

RCE-JEPA: Autonomous Deterministic Coding using Latent Space

David Charlot, Open Interface Engineering

Research paper · open topic · OpenIE research.openie.dev

<https://research.openie.dev/papers/rce-jepa/> · PDF <https://research.openie.dev/pdfs/rce-jepa.pdf>

Measurement. Research paper, open topic, not a journal final. No learned proposer has been trained; the Phase 0 results are checker-only measurements with 0 model calls. No joules were measured.

RCE is the Reality Code Engine, a deterministic programming system in which every program has one fixed, checkable shape. JEPA is a Joint Embedding Predictive Architecture, a kind of learned model that predicts a compact summary of the answer instead of writing the answer out word by word. Latent space is the space of those learned summaries: a list of numbers that keeps what matters about a program and drops surface detail. This topic asks whether software can be written autonomously by having a learned model propose programs in that closed, deterministic shape while a deterministic checker, not the model, decides what is accepted, with the best performance per clock cycle and per joule as the goal. Its thesis is that coding today resembles mechanics before Newton, rich in observation and craft but without rules that predict behaviour and cost in advance, and that a finite, deterministic, compressed program form makes such rules possible to state and test.

Status. This paper states a thesis, hypotheses, falsifiers and a programme of work; the results it reports are the Phase 0 measurements that need no learned proposer (the Form corpus, the name-free Form space, the determinism streams and the adversarial Forms, in the Status section), and no result for a learned proposer is reported because none has been trained. The deciding half (commit gate, exact replay, projection, energy accounting) is described in [the RCE paper](#); this paper is about the proposer: a learned model that steers search over Forms while the engine renders every target. Any model in the loop, hosted or local, only proposes, and security never depends on it.

 Pipeline: stated intent goes to a learned predictor working in latent space, a decoder proposes candidate program forms ranked by energy score, a deterministic checker accepts or names the broken rule, and accepted forms go to the engine for exact replay, proofs, code in other languages and placement

Figure 1. The proposed pipeline. A learned predictor maps stated intent to a point in latent space; a decoder turns that point into candidate programs in the fixed shape and ranks them by an energy score (lower means a better fit). A deterministic checker accepts or rejects each candidate and returns the broken rule for re-ranking. Everything inside the dashed box is learned and untrusted; only accepted programs reach the engine. Energy use is to be measured across the whole loop. [Open full size](#) · [PNG](#).

In plain words

1. People have tried for seventy years to get computers to write programs from a description. Each attempt fixed one part of the problem and left another part manual.
2. Today's AI (artificial intelligence) writes code one word piece at a time, in any shape, and often gets it wrong. A fixed program shape plus a strict checker turns "write anything" into "fill in a known shape, then get a yes or no".
3. The AI only proposes; the checker decides. So safety does not depend on the AI being right, and the cost that counts is the electricity spent per program that actually passes.
4. Before Newton, people could watch the planets but not predict them from a few rules. AI coding is in that position now. A program that always does the same thing from the same start, in a compact standard form, could be predicted, and its energy cost counted, before it runs. That is a conjecture, and the tests are listed below.

Principles

- The model only proposes.
- The checker decides.
- Security never depends on the model.
- No model runs inside the Reality Code Engine's own toolchain loop.
- Every accepted program form is replayable bit for bit.
- Any decision model, hosted or local, is only a proposer outside the gate.

Terms and acronyms used on this page

Listed here so that each is defined before it is used. Acronyms that appear only inside reference entries (venues, report series) are not listed.

- AGX: NVIDIA's Jetson AGX line of embedded computers
- AILANG: a programming language for AI-written code (a project name)
- AIXI: Hutter's theoretical model of a universally intelligent agent
- ANDF: Architecture Neutral Distribution Format
- BLAKE3: a cryptographic hash function
- BLEU: bilingual evaluation understudy, a text-overlap score
- CALM: consistency as logical monotonicity
- CEGIS: counterexample-guided inductive synthesis
- CPU: central processing unit
- CUDA: NVIDIA's GPU programming platform
- DBOS: a durable-execution system (a project name)
- DIR: a directory path given to a command
- EB-JEPA: an energy-based JEPA library (a project name)
- EWD: the numbered manuscript series of Edsger W. Dijkstra
- ESBMC: efficient SMT-based bounded model checker
- FAIR: Meta's Fundamental AI Research group
- GPT: generative pre-trained transformer, OpenAI's model family

- GPU: graphics processing unit
- IBM: International Business Machines
- ICOT: Institute for New Generation Computer Technology (Japan)
- IEEE: Institute of Electrical and Electronics Engineers
- JACM: Journal of the ACM (Association for Computing Machinery)
- MDE: model-driven engineering
- MLIPS: million logical inferences per second
- MLIR: multi-level intermediate representation
- NAIL: a canonical-form programming language (a project name)
- NVIDIA: a maker of GPUs
- OMG: Object Management Group
- PAR: pattern-based automatic program repair
- PTX: NVIDIA's parallel thread execution instruction set
- RAND RM: a RAND Corporation research memorandum
- RAPL: Intel's running average power limit energy counters
- README: a repository's front-page documentation file
- RFC: request for comments, an Internet standards document
- RISC-V: an open reduced instruction set computer architecture
- RJ-F: label of a falsifier of the RCE-JEPA premise
- SAT: Boolean satisfiability
- SHA-256: secure hash algorithm with a 256-bit digest
- STAN-CS: Stanford computer science technical report series
- STOP: self-taught optimizer
- TOPLAS: ACM Transactions on Programming Languages and Systems
- UNCOL: universal computer-oriented language
- VAX: Digital Equipment Corporation's virtual address extension computers
- XML: extensible markup language

How this page is organized

The page is meant to be read top to bottom. After the hypotheses and falsifiers at a glance, it has these parts, in this order:

1. What already exists, and what this page is about (the proposer section)
2. The thesis, in five parts: Part 1, computation as a finite information system; Part 2, compression into Form and template; Part 3, kinematics; Part 4, a world model whose physics is known exactly; Part 5, consequences as testable predictions
3. Convergence by commitment
4. Section 1. History: seventy years of "say what you want, get the program"
5. Section 2. Why the earlier attempts stalled
6. Section 3. Current attempts
7. Section 4. Why now

8. Section 5. One concrete instance: the Reality Code Engine, with the hypotheses table and the falsifiers
 9. Status and references
-

Hypotheses at a glance

The page tests thirteen hypotheses, H1 to H13, listed here in full before any of them is discussed. Each is stated with its metric and thresholds in one table in [Section 5](#), followed by the falsifiers RJ-F1 to RJ-F5.

1. H1. Form space is much smaller than source space for the same programs.
 2. H2. The checker is cheap relative to generation.
 3. H3. Prediction in representation space reaches useful acceptance.
 4. H4. Energy per accepted program falls.
 5. H5. Energy ranking concentrates mass on valid Forms.
 6. H6. The reference apps are encodable as Forms.
 7. H7. Security does not depend on the model.
 8. H8. Every accepted Form replays deterministically.
 9. H9. Cost is predicted before execution.
 10. H10. Learning on the quotient (canonical Forms) saves training data.
 11. H11. Energy is monotone in edit distance.
 12. H12. The decision head is calibrated on gate outcomes.
 13. H13. A discrete latent plans better than a continuous one over long edit sequences.
-

Falsifiers at a glance

Five observations that would defeat part of the premise, stated in full in Section 5 after the hypotheses they test.

1. RJ-F1. Lift coverage plateaus below 80% on any of the five target languages.
 2. RJ-F2. Writing the intent specifications for the reference applications costs more than writing their Forms by hand.
 3. RJ-F3. H3 fails while the token arm passes at the same gate.
 4. RJ-F4. H11 fails and search does not reach accepted Forms within the beam budget.
 5. RJ-F5. The decision head's calibration never beats a frequency table over shape hashes.
-

What already exists, and what this page is about

The Reality Code Engine already holds the half of the problem that decides. Its public paper, [The Reality Code Engine](#), describes the design of that half: a commit gate that checks every candidate at entry, exact replay (the runtime state is a left fold of the transition function over the log), projection of one Form to other languages, and energy accounting in `est_j`, an

estimate in joules from a published cost table, kept apart from counter readings (reported_j) and wall-meter readings (measured_j). Some theorems about it are checked in Lean 4, a proof assistant, and the specification and Lean library are public. The measured runs reported in that paper are not all reproducible from the public repositories yet, so this page quotes no number from them and treats them as design context, not as evidence for any hypothesis below.

This page is about the proposer: the learned part that suggests which Form to try. The gate, replay, projections and est_j are built and measured; the proposer is not.

The model steers; the engine renders

Robotics already works this way. In V-JEPA 2-AC, the action-conditioned version of the V-JEPA 2 video model, a planner searches over robot actions by minimizing the distance between a predicted future representation and the representation of a goal image; it never generates pixels [B1]. The encoder was pretrained on more than one million hours of video, and the action-conditioned predictor on fewer than 62 hours of unlabeled robot video [B1, abstract]. RCE-JEPA proposes the same division for code: a learned model chooses edits to a Form, and the engine renders every target language from the accepted Form. No model emits code.

Code differs from robotics in one decisive way. A robot's world model must *learn* how the world moves, $z_{t+1} = g_\phi(z_t, a_t)$. In RCE the transition function δ is given and proved, so the forward dynamics need not be learned at all. What remains to learn is the inverse: which edit moves a candidate toward what the intent requires. MuZero learns a model because, as its authors put it, planning methods "rely on knowledge of the environment's dynamics" [T1, Sec. 1]; here those dynamics are written down.

One real Form

Before the architecture, the object. This is a complete program from the engine repository (crates/rce/examples/surface/counter.rce), fourteen lines in RCE's surface syntax, shown unedited:

```
intent "a counter that never goes negative"
  example inc, inc, dec -> 1
  example dec -> refused: Underflow

state Counter
  n: u64 = 0

event Inc
event Dec

error Underflow
```

```

step Counter on Inc -> n + 1
step Counter on Dec -> try n.checked_sub(1) else Underflow

```

Read it as $P = (S, E, \delta, D, \lambda)$. The state block is S , one unsigned 64-bit integer starting at zero. The two event lines are E . The two step lines are δ . The intent block is a small specification of intent, written $\sigma(\iota)$ on this page: two required traces, one of which must be refused. A decrement at zero is refused with `Underflow` and, by T8, leaves the state unchanged. The proposer's task, stated on this example: given only the intent block, find the state, event and step lines that the gate accepts.

Architecture

 Inside a dashed region labelled **learned and untrusted**: an intent encoder, a proposer acting as an inverse model, an exact rollout by the transition function, a Form encoder with energy, a tree search, and a decision head. Outside it: the gate, which alone admits programs, and the engine, which replays and projects. A rejection arrow returns the broken rule to the search

Figure 2. The model steers, the engine renders. Orange boxes are learned and untrusted; the blue box is the engine's exact rollout, used as a simulator during search; green boxes are built (see the RCE paper for their status). Only the gate admits. A design diagram, not data. [Open full size](#) · [PNG](#).

Component	Role	Starting design	Why
Form encoder E_f	Map a Form and its trace on example events to a latent z_f	A small transformer over the canonical JSON (JavaScript Object Notation) of the Form plus trace tokens, regularized with SIGReg (sketched isotropic Gaussian regularization) [B2]; a second arm with a discrete or soft-codebook bottleneck [B3, B4, B5] (SC-JEPA's soft codebook is stabilized together with an EMA (exponential moving average) teacher and a reconstruction anchor [B5, Sec. 4.4.1 and App. B.1])	Traces define equivalence, and the trace is free because δ is exact. Discrete tokens kept long-horizon prediction stable where continuous latents accumulated error [B3]; Boolean latents allow exact state re-identification by bitwise equality, a motivation [B4] states with reference to earlier work
Intent encoder E_ι	Map $\sigma(\iota)$ (required traces, contracts, tests, prose) into the same space	Shared architecture, trained so that $E_\iota(\sigma(\iota))$ lies near $E_f(f^*)$ for known accepted fits f^*	Puts goal and candidate in one metric, as V-JEPA 2-AC does with goal images [B1]
Proposer Π_θ	Given z_ι and the current z_f , propose a typed Form edit	A goal-conditioned inverse model, the shape LeWAM compares against as GC-IDM (goal-conditioned inverse dynamics model), a baseline from	With δ exact, the inverse is the learnable object

Component	Role	Starting design	Why
		Nguyen et al. (2026) that LeWAM evaluates, not a model proposed by LeWAM's authors [B6, Sec. 4.2 and App. A.2]	
Energy $\mathcal{E}$	Rank candidates before the gate	$\alpha V(f) + \beta d(z_{\tau(f)}, z_{\sigma(i)}) + \gamma C(f)$, with $C(f)$ taken from <code>est_j</code>	Separately trained energies are not commensurate [T2, Sec. 1.3]; α, β, γ are fitted on a held-out slice to maximize AUROC (area under the receiver operating characteristic curve) for accept against reject, and reported with every run
Planner	Search over edits	Beam or tree search over proposer edits, scored by $\mathcal{E}$, with exact rollout at every node	V-JEPA 2-AC plans with the cross-entropy method (CEM) over continuous actions [B1]; Form edits are discrete and combinatorial, so CEM does not apply
Decision head	Bounded selection with a calibrated probability: the value for each hole, a rank, an equivalence verdict	A small decision model built by the RCE project and trained on gate receipts (next subsection)	The calibrated probability p_i orders the cheapest-first cascade

Not learned, ever: δ , λ , the fold, the hash chain, the projections, the gate and the promotion path. They are fixed and certified, and promotion never retrains them.

The decision head is a proposer, and it sits outside the gate

A class of small, energy-aware decision models appeared in 2026. A decision model takes a piece of state and a fixed question and returns a typed answer with a probability for every option, without generating text; all questions about one state are answered in a single request [B7; B8]. Answers are bounded: a yes or no, a choice among a caller-supplied set of options (up to 255 for Jev; Laya's accuracy drops beyond about 20), or a score on a declared scale [B8]. Members of the class, cited here as convergent examples of the concept and not as components of RCE:

- **Laya**, an open-weight model of about 421 million parameters built on a ModernBERT-large encoder with a two-layer decision head, released under Apache 2.0 [B8, B9].
- **Kev**, a family of open decision models from 0.8 to 27 billion parameters built on Qwen backbones, each checkpoint shipped with a fitted temperature for calibration [B10].

- **Liquid AI's open d1 models**, d1-3B and d1-omni-600M, released on 7 October 2026 for edge and physical systems: d1-3B answers one question in 16 ms on an NVIDIA Jetson AGX Thor and 50 ms on a Jetson Orin Nano, reads camera frames, and was shown navigating an environment in the Isaac Sim robot simulator with the model served on a Jetson in a hardware-in-the-loop setup [B11].

An independent benchmark of one hosted member of the class found its choice probabilities well calibrated, with an expected calibration error (ECE) of 0.028 pooled over 22 choice datasets and 279,925 answers [B7]; its binary (yes or no) probabilities were less well placed (ECE 0.052 single, 0.168 multi-label) and needed tuned thresholds [B7, Sec. 4.3]. Documented weaknesses of the class include counting, arithmetic and date comparison, and lower accuracy when the state is padded with irrelevant content [B8, B12].

What the RCE project plans. The project will build its own decision head, trained and calibrated on its own gate receipts. Every gate decision already writes a receipt with the candidate, the outcome, the broken rule and `est_j`, so calibration data costs no labelling. The state handed to the head is the Form's canonical JSON and its trace, which is small by construction; that addresses the documented padded-state weakness directly. The head is a proposer. Any decision model, hosted or local, built by the project or by anyone else, only proposes; it never admits a Form, and the security of RCE never depends on it.

Data

V-JEPA 2 pairs very large passive pretraining with a small action-conditioned set [B1]. The RCE analogue has four tiers.

Tier	Source	Size target	Trains
Passive: Forms and traces	Forms lifted from pinned public corpora by the engine's lift pipeline (<code>rce scan DIR --form</code>), the seeded 10,000-Form generator, and every promoted handler	10^5 to 10^6 Forms with traces	The Form encoder, by predicting masked rules and masked trace segments in latent space
Paired: intent and fit	Tests and contracts attached to lifted programs; a Form's own trace as a self-generated specification; the preregistered brownfield applications; benchmark specifications written as Forms	10^4 pairs to start	Intent encoder and proposer
Decision receipts	Every gate decision	Unbounded, no labelling cost	The decision head's calibration
Hard negatives	Gate rejections with the named broken rule; model-proposed equivalences the gate refused (13 of 16 from Qwen2.5-Coder-1.5B in the RCE paper, Sec. 5.5)	Grows with every run	Contrastive signal: valid against invalid, equivalent against not

Near-duplicate Forms are removed by shape hash before any split, and held-out splits are by shape hash, never by surface text. The first corpus exists: version 1 of the Form corpus (23 October 2026) holds 192 distinct shapes of hand-written programs from the public repository (version 4, 24 October 2026), 55 of them with an intent block, and 3,456 negatives (343 conformance rejects with their diagnostic code and 3,113 checker-verified mutants, whitespace-token and typed, refused across 31 diagnostic codes), with a dataset card and a split by shape hash; its generator is `scripts/corpus.py` in the repository and the rows are at research.openie.dev/data/. It is the paired and hard-negative tiers at their smallest; the passive tier (lifted programs) is empty until the lift pipeline is public. On it, H1 has a first measurement, preregistered before the run: the median of $\rho = b(\text{code})/b(P)$ over 185 programs is 9.34 against the emitted Python (interquartile range 8.45 to 10.81; version 1 gave 9.31 over 173, with 17.33 for Rust, 11.06 for JavaScript and 13.66 for Scala), above the threshold of 5; 171 of the first 173 programs were above 5 (Wilson 95% interval 0.96 to 1.00). Under a compressed description length (gzip) the ratio is 3.57 for Python and 5.73 for Rust, which is reported next to the primary so that the number is not read as a claim about information content: H1 is about the space a proposer searches, which is the token count under its vocabulary. The emitted source is the engine's own, with the shared runtime excluded, and no lifted or model-written program is in the population. The same run gives two further numbers the hypotheses need. The checker side of H2: `rce check` takes a median of 3.0 ms per program (interquartile range 2.6 to 3.8 ms), about 0.03 J at a stated 10 W, labelled an estimate; the model side is not measurable without the proposer. And a measure of what an intent block buys, which RJ-F2 and H7 both rest on: of the mutants that change a program's behaviour, its own example lines catch 317 of 392 (0.81, Wilson 95% interval 0.77 to 0.85; 24 of 54 programs catch every one), and the uncaught changes are almost all a constant moved by one that no example exercises.

Training, in brief

- **Encoder objective.** Latent prediction plus SIGReg, the recipe of LeJEPA, which needs a single trade-off hyperparameter and no stop-gradient or teacher-student scheme [B2, abstract], and of LeWorldModel, which trains end to end with two loss terms [B13, abstract]. Two cautions from the same line of work: the LeWorldModel authors flag the isotropic Gaussian target as a possible limitation in environments of low intrinsic dimension [B13, Sec. 4.2 and Sec. 6], and contrastive inverse dynamics is offered as a distribution-free alternative [B14], which, however, was evaluated on continuous controls only and was clearly worse than SIGReg on combinatorial puzzle tasks whose discrete state changes look alike whichever action caused them [B14, Appendix D]; and SIGReg and VISReg "do not guarantee that the learned latent representation preserves all the key information required for planning" [B15, Sec. 4.1]. Both arms run side by side, with auxiliary heads that predict exact, cheap properties of the Form (monotone tag, fuel bound, rule count).
- **Multi-step prediction.** Training with k-step recursive prediction gives significantly better results in the EB-JEPA ablations, by reducing exposure bias, with a Pareto optimum near $k = 4$ [B16, Fig. 3]; RoboJEPA raises the autoregressive training rollout from $K = 2$ to $K =$

10 steps in a long-horizon cooldown, which pays off only at large compute (above about 10^{21} FLOPs) [B17, Sec. 3.3 and App. C.2]; a rollout-consistency term matched to the planner horizon comes from Temporal-Distance JEPA [B18, abstract].

- **Monotone energy.** A monotone cost ranking makes plans that deviate further from a goal-reaching trajectory cost more; without it, local minima in the cost surface can lead the planner to suboptimal plans [B19]. Monotonicity is not guaranteed by the task: the EB-JEPA navigation task is non-monotonic, since the best path first moves away from the goal [B16]. Here it is a preregistered diagnostic (H11), not an assumption.
- **Proposer.** Supervised on intent, Form and edit triples from the paired tier, then improved on its own gate-accepted trajectories: the gate is the reward.
- **Never decode a latent directly.** MoRAE reports that generating motion directly in a frozen JEPA latent "fails dramatically", and traces the failure to an ill-conditioned latent spectrum and to small latent errors amplified into large artifacts after decoding [B20, abstract and Sec. 3]. The proposer outputs typed edits; latents rank, they do not render.

Convergence: where each piece already exists

No single piece of RCE-JEPA is new. Each has an existing home, listed below as convergence, not priority. What has not been measured is the joined system, which is the subject of the hypotheses.

Piece	Where it exists	What it established	Difference from RCE-JEPA
Predict a compact plan, then decode the program	Latent Programmer [B21]	A discrete latent code predicted from the specification, then the program; string-transformation accuracy up 5 to 7 points (over 10% relative) over the best baseline, plus higher BLEU on Python code generation from docstrings [B21, Sec. 5.1 and 5.3]	Open target language, no canonical form, no checker
Learn execution, not only text	LaSynth [B22]; Code World Model (CWM) [B23]; analysis of CWMs [B24]	Latent execution improves synthesis in a restricted C domain; CWM is mid-trained on observation-action trajectories from a Python interpreter and Docker environments	Dynamics are learned; in RCE they are exact
Equivalence classes as a learning target	EqNet, Allamanis et al. [B25]	Continuous representations of symbolic expressions trained on equivalence classes	Expressions, not whole programs
Agents propose, solvers verify	Agentic model checking [B26]; verified synthesis for concept design driven by a large language model (LLM) [B27]; AlphaProof, Clover,	"Agents propose, solvers verify" is stated as a principle; CEGIS-style synthesis (counterexample-guided inductive synthesis) against Alloy invariants and positive and negative scenarios	Proposers generate tokens; no latent planning, no joule accounting

Piece	Where it exists	What it established	Difference from RCE-JEPA
	AutoVerus (Section 3 below)		
JEPA as a latent reasoner	JEPA-Reasoner [B28]	Latent reasoning decoupled from token generation	Text domain
Cost predicted before execution	Compiler world models [B29]	A transition model rolls out scheduling actions in latent space to predict the terminal-state representation, which a ranker uses to order candidates before fewer hardware measurements; the score is a within-workload rank, not a calibrated cost [B29, Sec. 3.2 and Sec. 6]	Tensor program schedules only
Discrete latents	Discrete-JEPA [B3]; discrete world models via regularization [B4]; SC-JEPA [B5]	Discrete tokens that keep long-horizon symbolic prediction stable [B3]; Boolean latents learned without reconstruction (planning left to future work) [B4]; a soft codebook bottleneck, with an EMA teacher and a reconstruction anchor, to stabilize one-step latent prediction, not long-horizon rollout [B5]	Synthetic images, puzzles, time series; not programs
Stable JEPA training	LeJEPA [B2]; LeWorldModel [B13]	Collapse-free training without stop-gradient, teacher networks or hyper-parameter schedulers; a standard learning-rate warm-up and cosine schedule is still used [B2, abstract and Sec. 6.1]; without stop-gradient, EMA or pretrained encoders, with two loss terms [B13, Sec. 3.1]	Reusable as an objective; validated only on images [B2] and pixels [B13], untested on program data
Small calibrated decision models	Laya [B9]; Kev [B10]; Liquid AI open d1 [B11]; benchmark [B7]	Typed answers with per-option probabilities. Kev ships a fitted temperature per checkpoint [B10]; Laya ships over-confident (mean ECE 0.466) and reaches 0.081 only after a per-type temperature refit [B9]. Latency: about 33 to 40 ms per question on a T4 GPU for Laya and 18.1 ms for six questions on an H100 for Kev, but hundreds of milliseconds on a CPU or Mac (Laya 193 to 464 ms on CPU, Kev-4B 721 ms on an M5 Mac) [B9, B10]; d1-3B answers one question in 16 ms on a Jetson AGX Thor and 50 ms on a Jetson Orin Nano [B11]; one hosted member independently benchmarked for calibration [B7]	No world model; in RCE-JEPA a decision head is one proposer among several

The specification cost, named

The checker guarantees that a Form is valid; it does not guarantee that it is the Form that was meant. In verified LLM-driven synthesis for concept design, invariant-only synthesis "reached verified designs quickly but often produced inconsistent designs across runs", which shows "that invariants alone underconstrain the design task"; scenario-guided synthesis recovered intended designs more consistently than natural-language prompting, although minimal scenarios could overfit [B27, abstract], on 3 applications and 12 variants with one LLM configuration [B27, Sec. 5]. The dominant failures "were not failures of bounded verification, but failures of specification coverage" [B27, Sec. 5.6]. In seL4 the abstract specification took about 4 person-months and the proof about 20 person-years (Klein et al. 2009, Section 2 below). A closed program world makes acceptance exact and search tractable. It moves the uncertainty of intent into $\sigma(\iota)$; it does not remove it. Falsifier RJ-F2 below makes that cost public and measurable.

Proposer hypotheses

The proposer is tested by H11 (monotone energy), H12 (decision-head calibration) and H13 (discrete against continuous latents). All thirteen hypotheses, H1 to H13, are stated in numerical order with their thresholds in one table in Section 5 below, under the same conventions: preregistered before the run, frozen engine commit and binary hash recorded, same hardware for every arm.

The H3 control is fixed now: the same gate and the same `est_j` accounting, with Qwen2.5-Coder-1.5B proposing surface code, the arm already run in the RCE paper (Sec. 5.5), where the gate admitted 3 of 16 proposed equivalences.

Falsifiers. The falsifiers for the premise itself, RJ-F1 to RJ-F5, now follow the hypotheses they test, directly after the H1 to H13 table in [Section 5 below](#).

Risks and mitigations

Risk	Evidence	Mitigation
Decoding a latent amplifies small errors	Direct generation in a frozen JEPA latent fails dramatically for motion, from spectral ill-conditioning and decoder amplification [B20]	Never decode a latent; the proposer emits typed edits and the gate checks them
A representation that does not collapse but learns the wrong thing	"Physical representation laziness" under SIGReg [B15]	Auxiliary exact-property heads; an inverse-dynamics regularizer
Isotropic prior misfits a low-dimensional Form space	[B13]; [B14]	SIGReg and contrastive inverse-dynamics arms side by side, remembering that the inverse-dynamics arm was weaker on discrete puzzle state [B14]

Risk	Evidence	Mitigation
Energy terms not commensurate	[T2, Sec. 1.3]	Fit α, β, γ per experiment and publish them
Decision head degrades on padded state	Documented weakness of the class [B8, B12]	Feed only canonical Form JSON and trace
Expressiveness ceiling	Rice's theorem [T3]; the restricted language	Lift coverage is the measurement; RJ-F1 is the falsifier
Specification cost dominates	seL4 effort (Section 2 below); invariant under-constraint [B27]	RJ-F2; tools that capture traces from running programs and infer contracts, as deliverables in their own right

The phased pilot plan, from the data engine through convergence, with preregistration and exit criteria for each phase, is kept with the project's working notes and summarized here: Phase 0 builds the lift corpus and receipt dataset; Phase 1 trains the Form encoder and tests H1, H5 and H11; Phase 2 builds and calibrates the decision head (H12); Phase 3 trains the proposer and planner against the token control (H3, H4, H10, H13); Phase 4 measures energy and determinism on a wall meter (H2, H8, H9); Phase 5 runs the convergence protocol, in which promotion takes the model out of the loop for each closed shape. Alongside Phase 0, a slice of the vericoding benchmark (source C34, listed under Convergence by commitment below), chosen as its state-machine-shaped tasks in Dafny, Verus and Lean, is lifted into Forms and scored under RCE's gate next to the published vericoding results, an outside benchmark for the step from specification to Form (H6, H7 and H8).

Proposer references

References marked [T] are listed under Thesis references below; references to Sections 2, 3 and 5 are to the sections of this page below. Labels follow the convention of the main references below. **Full** here means the whole text was read, including references and appendices; for B9 and B10 the live model card and README were read on 14 October 2026.

- [B1] M. Assran et al., "V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning", 2025. [arXiv:2506.09985](#). Full.
- [B2] R. Balestrierio and Y. LeCun, "LeJEPA: Provable and Scalable Self-Supervised Learning Without the Heuristics", 2025. [arXiv:2511.08544](#). Full.
- [B3] J. Baek et al., "Discrete JEPA: Learning Discrete Token Representations without Reconstruction", 2025. [arXiv:2506.14373](#). Full.
- [B4] D. Bizzaro and L. Serafini, "Discrete World Models via Regularization", 2026. [arXiv:2603.01748](#). Full.
- [B5] Y. He et al., "SC-JEPA: Stabilizing Latent Predictive Learning for Time-Series Anomaly Prediction", 2026. [arXiv:2602.04643](#). Full.
- [B6] M. Fu et al., "LeWAM: End-to-End World and Action Modeling with JEPAs", 2026. [arXiv:2610.12818](#). Full.

7. [B7] T. Deußer, L. Sparrenberg and R. Sifa, "Evaluating and Benchmarking the System One Model Jev", 2026. [arXiv:2609.37647](#). Full.
8. [B8] W. Wu, "Jev vs Laya: Comparing and Choosing Between Closed-Source and Open-Source System One Decision Models", 23 September 2026. [wilsonwu.me](#). Full.
9. [B9] Nandakishor M (Convai Innovations), Laya model card. [Hugging Face](#). Full (read live 14 October 2026).
10. [B10] J. Palmer, Kev. [GitHub](#). Full (README, read live 14 October 2026).
11. [B11] Liquid AI, "Open d1: Edge decision models for text, vision, and audio", 7 October 2026. [liquid.ai](#). Full.
12. [B12] Sanity, "Decision model" (glossary). [sanity.io](#). Full.
13. [B13] L. Maes, Q. Le Lidec, D. Scieur, Y. LeCun and R. Balestrieri, "LeWorldModel: Stable End-to-End Joint-Embedding Predictive Architecture from Pixels", 2026. [arXiv:2603.19312](#). Full.
14. [B14] J. Boylan and C. Hokamp, "No Gaussian Required: Contrastive Inverse Dynamics for JEPA World Models", 2026. [arXiv:2608.17542](#). Full.
15. [B15] P. Zhu et al., "Spectral-Target Physical Latent Structuring for JEPA-Style World Models", 2026. [arXiv:2609.04264](#). Full.
16. [B16] B. Terver et al., "A Lightweight Library for Energy-Based Joint-Embedding Predictive Architectures", 2026. [arXiv:2602.03604](#). Full.
17. [B17] A. Zholus et al., "RoboJEPA: Scaling Robotic Latent World Models", 2026. [arXiv:2610.10515](#). Full.
18. [B18] J. Bai and J. Xiong, "Temporal-Distance JEPA: Plan-Aware Representation Learning for Latent World Model Predictive Control", 2026. [arXiv:2607.25337](#). Full.
19. [B19] A. Chahe, S. Cai and L. Zhou, "Latent World Models with Monotone Planning Costs for Image-Goal Navigation", 2026. [arXiv:2608.09073](#). Full.
20. [B20] Y. Zhu et al., "MoRAE: Flow-Friendly Self-Supervised Latents for Text-to-Motion Generation", 2026. [arXiv:2607.29180](#). Full.
21. [B21] J. Hong, D. Dohan, R. Singh, C. Sutton and M. Zaheer, "Latent Programmer: Discrete Latent Codes for Program Synthesis", ICML 2021. [PMLR PDF](#). Full.
22. [B22] X. Chen, D. Song and Y. Tian, "Latent Execution for Neural Program Synthesis", NeurIPS 2021. [arXiv:2107.00101](#). Full.
23. [B23] Meta FAIR CodeGen Team, "CWM: An Open-Weights LLM for Research on Code Generation with World Models", 2025. [arXiv:2510.02387](#). Full.
24. [B24] B. Rahmani, "Debugging Code World Models", 2026. [arXiv:2602.07672](#). Full.
25. [B25] M. Allamanis, P. Chanthirasegaran, P. Kohli and C. Sutton, "Learning Continuous Semantic Representations of Symbolic Expressions", ICML 2017. [arXiv:1611.01423](#). Full.
26. [B26] Y. Sun, J. Liu, D. Kroening and J. Xue, "Agentic Model Checking", 2026. [arXiv:2605.21434](#). Full.
27. [B27] A. Cunha, "Verified LLM-Driven Synthesis for Concept Design", 2026. [arXiv:2607.15718](#). Full.
28. [B28] B. K. Liu, Z. P. Chen and D. P. Woodruff, "JEPA-Reasoner: Decoupling Latent Reasoning from Token Generation", 2025. [arXiv:2512.19171](#). Full.

29. [B29] H. Pan et al., "Toward Compiler World Models: Learning Latent Dynamics for Efficient Tensor Program Search", 2026. [arXiv:2606.09312](https://arxiv.org/abs/2606.09312). Full.
-

The thesis: software construction is pre-"Laws of Coding"

Picture Europe in the 1660s. Kepler had published two regularities of planetary motion in *Astronomia nova* (1609) and a third in *Harmonice mundi* (1619) [T4]. Galileo and Huygens had derived testable conclusions from hypotheses about motion [T5, Sec. 1]. Gunners, millwrights and shipwrights built by rule of thumb and long apprenticeship. The sky had been watched with enormous care for two thousand years, and yet prediction was a patchwork. Newton, by the account of George Smith, "was familiar with seven different approaches to calculating planetary orbits, all at roughly the same accuracy", and calculated positions "were sometimes off by as much as a fourth of the width of the Moon" [T5, Sec. 2]. There were many good tables and no single account of why the tables worked.

Then, in 1687, came the *Principia*. A handful of axioms of motion and one inverse-square force, and from them trajectories that could be computed in advance, costs that could be reckoned before the cannon was cast, and errors that pointed to new bodies rather than to bad luck. Smith summarizes the change: after the *Principia*, "physical theory gained primacy over observation for purposes of answering specific questions about the world" [T5, Sec. 1]. Galileo's results for uniform gravity did not disappear; they became a limit case of inverse-square gravity, and the theory went on to explain the planets' deviations from Kepler's rules [T5, Secs. 1 and 2].

The premise, in plain words: **the practice of software construction, including today's AI-assisted coding by large language models, is pre-"Laws of Coding"**. Here "the practice of software construction" means the practitioners and systems, human developers and code-generating models alike, that produce code by empirical trial, testing and review rather than by derivation from a formal rule set. That practice today lives in the 1660s. It has careful observation: benchmarks such as pass@k (the chance that at least one of k sampled programs passes the tests) and SWE-bench (a set of real repository issues), both described in Section 3 below. It has craft: prompt wording, retries, and trial and error. Outputs of code-generating models are samples from a probability distribution, so the same request can give a different program tomorrow; when ChatGPT (gpt-3.5-turbo at its default temperature) was asked the same coding question five times, for 47.56% to 75.76% of problems across three datasets there was no test on which all five answers gave the same output, and "setting the temperature to 0 does not guarantee determinism in code generation" (Ouyang et al., 2023, abstract; Section 3 below). The regularities that the human side of the practice does show are, in Lehman's words, ones that "arise from the habits and practices of people and organizations" [T6, Sec. V-A, p. 1069]. Hoare and colleagues named the same gap in 1987: "What then are the laws of programming, which provide the formal basis for the profession of software engineering? Many programmers may be unable to quote even a single law" [T7, p. 673]. Dijkstra had already marked the limit of the empirical method that the practice still relies on: "program testing can be a very effective way to show the presence of bugs, but is

hopelessly inadequate for showing their absence" [T8, Argument three]. There is no agreed conserved quantity for a program, no equation of motion that a code generator obeys, and cost is measured after the fact, in tokens and in electricity, rather than computed before the work begins.

"Laws of Coding" names the craft layer, defined with the other three layers below. The RCE-JEPA proposal, stated here as conjecture, is that this craft can gain a small set of rules from which the behaviour and the cost of the code it produces can be computed before it runs, by borrowing from the layers beneath it. The candidates are (L1) a deterministic transition rule, (L2) an account of information and energy per transition, (L3) a canonical Form for every program, which is compression, and (L4) a composition rule under which L1 to L3 survive when programs are joined. The [Reality Code Engine \(RCE\)](#) proposes candidates for each; hypotheses H1 to H13 are the tests, stated in numerical order in one table in Section 5 below. Nothing in this section is a measured result unless it carries a number and a source.

The analogy has limits, stated here. Newton described a world nobody designed; programs are made by people, so their rules are partly chosen rather than discovered. Mechanics has continuous quantities and exact conservation of energy; a program's state is discrete and finite, and the energy a chip spends depends on the hardware, not only on the program. Rice's theorem [T3] says that no general procedure decides any non-trivial property of what an arbitrary program computes, and the halting problem [T9] is the best-known case; RCE avoids these limits only by working in a restricted language whose programs carry termination bounds, and that restriction has a price in what can be expressed. Finally, even the *Principia* was not the "clockwork universe": Smith notes that aspect "was added by Laplace" [T5, Sec. 1]; the axioms of motion alone did not settle every problem of mechanics either, since for bodies that are not point-masses they "must be supplemented by further principles" [T5, Sec. 5]; and "Newtonian theory is now seen to hold only to high approximation in limited circumstances" [T5, Sec. 1]. The claim is an analogy, not an equivalence.

Four layers of law, from theory to art

Four phrases are easy to blur, and this page keeps them apart. Each names a different layer, running from theory through science to art, and each is formalized to a different degree today. They are never used interchangeably here.

A spectrum from theory to art with four separate layers: "Laws of Computation", "Laws of Programming Languages", "Laws of Programming" and "Laws of Coding", each with its question, its representative work and how formalized it is today, and RCE-JEPA drawn as a bridge across the layers rather than a merger

Figure 3. The spectrum. Left to right: "Laws of Computation" (theory), "Laws of Programming Languages" and "Laws of Programming" (science), "Laws of Coding" (art). The bar under each layer shows how formalized it is today, a qualitative judgement explained in the table below. RCE-JEPA is drawn as a bridge that carries evidence between layers; it does not merge them. A qualitative diagram, not data. [Open full size](#) · [PNG](#).

Layer	Definition	Who works on it	How formalized today
"Laws of Computation"	Physical and information limits: what can be computed at all, and at what cost in information, time and energy.	Shannon [T10]; Landauer [T11]; Bennett and Landauer [T12, T13]; Turing [T9] and Rice [T3]; Lloyd [T14]; Wolpert [T15]; Carette, James and Sabry [T16].	Exact theorems for limits: information in bits, the erasure floor $k_B T \ln 2$, undecidability. No unified account yet predicts the cost of a given program on a given machine; Bennett and Landauer note their limits are "terribly far away from the real limits of modern technology" [T13, p. 48].
"Laws of Programming Languages"	Semantics and type theory: the rules that fix what a program in a language means.	Hoare's axiomatic semantics [T17]; Milner [T18]; Harel [T19]; Lampert [T20]; Berry and Gonthier [T21]; Halbwachs et al. [T22]; Leroy's verified compiler and Letouzey's extraction (Section 1 below).	Fully formal for some languages and compilers, machine-checked in CompCert (Section 1 below). Formal meaning is a property a language is designed to have, not one every language has.
"Laws of Programming"	Algebraic laws of correct program construction: equations by which programs are transformed and derived from specifications.	Hoare, Hayes, He, Morgan, Roscoe, Sanders, Sorensen, Spivey and Sufrin [T7]; Dijkstra's derivation of programs and Bird's calculus of lists (Section 1 below).	Nearly one hundred laws for a small language [T7]; by its authors' own account "still a long way from knowing how to apply them directly to the design of correct and efficient programs on the scale required by modern technology" [T7, p. 686].
"Laws of Coding"	The empirical craft of producing working code by people or models. This is where AI coding sits today.	Brooks [T23]; Lehman [T6]; Hindle et al. [T24]; the benchmark builders of Section 3 below (pass@k, SWE-bench).	Empirical regularities and benchmarks, no laws. Outputs of AI coders are samples, so the same request can give a different program (Section 3 below).

How RCE-JEPA relates to the layers. RCE-JEPA is a bridge, not a merger. It works in the "Laws of Coding" layer, where a learned model proposes code, and it ties each proposal to the layers beneath: the Form has a fixed meaning ("Laws of Programming Languages"), accepted Forms can be transformed by stated rules ("Laws of Programming"), and their energy is accounted against physical limits ("Laws of Computation"). Each hypothesis below names the layer it tests. If the bridge holds, the craft gains rules borrowed from the science and the theory; the layers stay distinct.

Mechanics has its *Principia*. Computation does not, yet. The pieces of "Laws of Computation" listed above are exact, but they have not been joined into one formal account that predicts behaviour and cost together, the way the *Principia* joined Kepler's and Galileo's regularities.

This is stated as an open problem, not a solved one.

Nothing in that gap is a failure of the people who came before. Each era did what its machines, its theory of information and its craft of building and optimizing programs allowed. The timeline below says what each era had to work with, with figures from the sources named, and what it achieved.

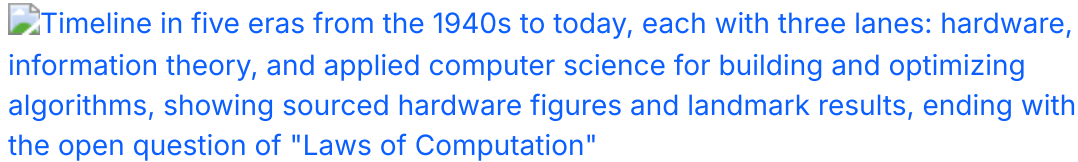
Timeline in five eras from the 1940s to today, each with three lanes: hardware, information theory, and applied computer science for building and optimizing algorithms, showing sourced hardware figures and landmark results, ending with the open question of "Laws of Computation"

Figure 4. Five eras on the road toward "Laws of Computation". Top lane: hardware, with figures from the sources in the table below. Middle lane: information theory. Bottom lane: applied computer science for developing and optimizing algorithms (compilers, complexity, synthesis, verification). The right-hand box is the open question: a unified "Laws of Computation". A historical diagram, not data. [Open full size](#) · [PNG](#).

Era	Hardware (memory, operations, energy)	Information theory	Applied computer science for algorithms	What it made possible, and where it stopped
1940s to 1950s	Single-copy machines whose memory "was usually too small" and which were "usually too slow" [T8]. On the IBM 704, a 47-statement FORTRAN (FORMula TRANslation) program compiled "in six minutes, producing about 1000 instructions" [T25, PDF p. 2].	Shannon (1948) makes information a measurable quantity: "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point" [T10, Introduction, p. 1].	The first optimizing compilers [T25].	Translation from formula to machine code became automatic. Every instruction and every word of memory was dear, so effort went to squeezing programs into machines, as Dijkstra recalls [T8].
1960s	The four models of IBM System/360 (1964) differed by a factor of 16 in memory capacity and 50 in performance while sharing one instruction set [T26].	Landauer (1961) ties erasure of information to heat [T11]; Solomonoff, Kolmogorov and Chaitin define complexity as shortest description [T27, T28, T29].	Hoare's axiomatic basis for proving programs correct (1969) [T17].	Correctness and information got exact definitions on paper. Dijkstra (1972): as "the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in

Era	Hardware (memory, operations, energy)	Information theory	Applied computer science for algorithms	What it made possible, and where it stopped
				proportion" [T8], and the software crisis followed.
1970s to 1980s	Alto (1973): a writable control store of 4,096 words of 32 bits; VAX-11/780 (1977): a control store of 5,120 words of 96 bits; Moore's 1975 projection of a doubling in transistor density every two years [T26].	Bennett (1973) shows that general-purpose computing machines "may be made logically reversible at every step, while retaining their simplicity and their ability to do general computations" [T30, abstract, p. 525]; Bennett and Landauer (1985) set out the physical limits [T13].	Brooks (1986) separates essential from accidental complexity [T23]; Hoare and eight co-authors give nearly one hundred algebraic laws for a programming language [T7]; Lehman states empirical laws of software evolution [T6].	Programs became algebraic objects. Hoare et al. are candid about the distance left: "we are still a long way from knowing how to apply them directly to the design of correct and efficient programs on the scale required by modern technology" [T7, p. 686].
1990s to 2000s	Instruction-level parallelism and faster transistors gave about 50% annual performance growth from 1986 to about 2002 [T26].	Lloyd (2000) bounds operations per second by energy [T14].	McKee (1995) proposes fundamental laws of computation [T31]; machine-checked compilers and kernels, CompCert and seL4 (Section 1 below).	Proof reached real systems, at a price: seL4's proof took about 20 person-years (Section 1 below).
2010s to today	Dennard scaling "faded to almost nothing by 2012"; a 32-kilobyte cache access costs about 200 times the energy of a 32-bit add, and off-chip DRAM (dynamic random-access memory) about 10 times a last-level cache access; Google's TPU (Tensor Processing Unit) v1 performs 256 x 256 multiply-accumulates per cycle with 24 megabytes of local memory, at more than 80	Wolpert's stochastic thermodynamics of computation [T15]; prediction shown equivalent to compression for language models [T32]; Carette, James and Sabry rebuild models of computation on conservation of information [T16].	Learned code generators; model plus verifier loops; deterministic simulation testing (see who holds these views, below).	Checking is cheap and exact where proposers are cheap and plentiful. A Python matrix multiply rewritten for the hardware ran "more than 62,000x faster" [T26]: the distance between a program and its machine is now

Era	Hardware (memory, operations, energy)	Information theory	Applied computer science for algorithms	What it made possible, and where it stopped
	times a CPU's energy efficiency on its workload [T26]. A 65-billion-parameter language model used about 3 to 4 joules per output token (Samsi et al., Section 4 below).			large, measurable and worth closing.

Why progress is possible now (opinion, to be tested). Three things that were never available together are available today. Machine-checked verifiers exist for real systems, so a rule can be enforced rather than hoped for. Learned models can propose candidate programs by the thousand, so the search for a program that satisfies a rule is affordable. And energy can be read from counters on ordinary hardware, so cost can be measured against prediction rather than guessed. Each earlier era lacked at least one of the three. None of this makes "Laws of Computation" inevitable; it makes them testable, which is the hypotheses' purpose below.

Symbols used in this section

- $P = (S, E, \delta, D, \lambda)$: a program as a functional state machine. S is a finite set of states; E a finite set of events (inputs); $\delta : S \times E \rightarrow S$ the transition function; D the set of derivations, values computed from a state by pure functions; $\lambda : S \times E \rightarrow O$ the output function into a finite output set O .
- Ω_n : the set of canonical Forms of size at most n symbols. A Form is the checker-accepted canonical representation of a program in RCE (Section 5).
- X : the set of surface programs (the text people or models write). $\kappa : X \rightarrow \Omega$: the canonicalization map from surface program to Form.
- $H(\cdot)$: Shannon entropy in bits [T10]. $K(x)$: Kolmogorov complexity, the length in bits of the shortest program that outputs x on a fixed universal machine [T27, T28].
- $k_B = 1.380649 \times 10^{-23} \text{ J K}^{-1}$: Boltzmann's constant. T : temperature in kelvin (K). J : joule.
- JEPA: Joint Embedding Predictive Architecture [LeCun 2022, cited above]. MDL: minimum description length [T33]. EBM: energy-based model.

Part 1. Computation as a finite information system

Strip a program of its syntax, its comments, its indentation and its author's habits, and what remains is a machine that moves from one state to the next, one event at a time. That machine is finite, and a finite thing can be counted.

Definition. A program is a deterministic functional state machine $P = (S, E, \delta, D, \lambda)$ with S and E finite and δ a total function. Determinism means δ is a function, not a relation: one state and one event give exactly one next state.

Bound on information content. If a Form is drawn from Ω_n by any distribution, its entropy is at most that of the uniform distribution: for n outcomes, "H is a maximum and equal to $\log n$ when all the p_i are equal" [T10, Sec. 6, property 2, p. 11]:

$$H(P) \leq \log_2 |\Omega_n| \text{ bits.}$$

For an alphabet of a symbols, $|\Omega_n| \leq \sum_{k \leq n} a^k$, so $H(P) \leq n \log_2 a + 1$ bits. Every Form, however long its behaviour, is a finite message.

Compression and understanding. Kolmogorov complexity $K(x)$ [T27, T28] measures the shortest description of an object. Kolmogorov takes as the "relative complexity" of an object y given x "the minimal length $l(p)$ of the 'program' p for obtaining y from x " [T28, Sec. 3, p. 165]; Solomonoff's induction [T27] and Rissanen's MDL [T33] turn "prefer the shortest description" into a rule for prediction and model choice. The bridge to learning is Shannon's source coding theorem [T10, Sec. 9, Theorem 9, p. 16]: a good code spends about $-\log_2 p(x)$ bits on x (Shannon's second proof gives a message of probability p_s a code of m_s binary digits with $\log_2(1/p_s) \leq m_s < 1 + \log_2(1/p_s)$ [T10, Sec. 9, p. 17]), so a better predictor is a better compressor. Delétang et al. state it directly: "maximizing the $\log_2$ -likelihood (of the data) is equivalent to minimizing the number of bits required per message" [T32, Sec. 1]. K itself is not computable: Kolmogorov already noted that the conditional complexity "cannot be effectively calculated (generally recursive) even if it is known to be finite for all x and y " [T28, Sec. 3, p. 165], and Schmidhuber notes "there is no general way of finding the shortest program computing any given data" [T34, Sec. 1]. In a restricted language with a canonical Form, the Form length is a computable upper bound on K up to an additive constant.

Part 2. Compression into Form and template

Ask a hundred programmers to write a shopping cart and you will get a hundred programs. Underneath the hundred there are perhaps three or four ideas. The surface is a crowd; the Form is the face it shares.

Equivalence classes. Define $x \sim y \iff \kappa(x) = \kappa(y)$. Each Form $f \in \Omega$ is the representative of the class $[f] = \kappa^{-1}(f)$ of $m_f = |[f]|$ surface programs. Because κ is a function, the chain rule of entropy, $H(x, y) = H(x) + H_x(y)$ [T10, Sec. 6, property 5, p. 12], gives

$$H(X) = H(\kappa(X)) + H(X | \kappa(X)).$$

A predictor that outputs the Form rather than the surface text is spared the second term. If surface variants within a class are equally likely, the saving per Form is

$$\Delta H_f = \log_2 m_f \text{ bits,} \quad \rho_f = \frac{|x|}{|f|} \quad (\text{bytes of surface code per byte of Form}).$$

This is why the predictor works on the quotient space $X/\sim \cong \Omega$: it learns a smaller, cleaner target, and every point in it is something the checker can judge.

Templates as low-dimensional manifolds. A template or a no-code schema is a Form with holes. If a template has h holes each taking one of v values, it covers v^h Forms and is addressed by $h \log_2 v$ bits, typically far fewer than $\log_2 |\Omega_n|$. Program space is vast; the programs people actually want lie on thin sheets within it. This is a conjecture, consistent with the observation that code is more predictable than natural language [T24] and with learned code embeddings in which "semantically similar objects are mapped to close vectors" [T35, Sec. 1].

Corpus measurements. No measurement of the compression ratio ρ or of the saving ΔH is reported on this page. They are planned: the evidence for H1 is to be measured on canonical serialization with shape-hash deduplication over the lift corpus described in the proposer section above. Until that run exists, H1 is a hypothesis with no result.

Part 3. Kinematics: a Newtonian form of computing

Throw a stone and, given where it starts, how fast, and what pushes on it, Newton tells you where it lands. A deterministic program has the same character, and more strictly: no air, no wind, no rounding of the world. Given where it starts and what events arrive, its entire future is fixed.

Trajectory. For initial state s_0 and event stream $e_0, e_1, \dots, e_{T-1}$,

$$s_{t+1} = \delta(s_t, e_t), \quad t = 0, \dots, T-1.$$

The correspondence proposed here, as analogy:

- state s_t plays the part of position;
- an event e_t plays the part of an applied impulse;
- δ plays the part of the equation of motion;
- the event log is a record from which the trajectory can be replayed exactly, which is reversibility in the record even when δ discards information; Bennett shows that "it is possible to reprogram any deterministic computation as a sequence of logically reversible steps, provided the computation is allowed to save a copy of its input" [T12];
- step count or fuel t plays the part of time;
- energy per transition, in joules, is the cost.

Energy floor. Landauer argued that logical irreversibility "is associated with physical irreversibility and requires a minimal heat generation, per machine cycle, typically of the order of kT for each irreversible function" [T11, abstract, p. 183]; resetting a bit cuts the entropy by $k \log_e 2$ and so supplies "0.6931 kT per restored bit to the surroundings" [T11, Sec. 4, p. 187]. Landauer's principle, as Bennett later restates it, holds that a logically irreversible operation such as erasure "must be accompanied by a corresponding entropy increase" in the environment [T12, abstract]. The floor per erased bit is

$$E_{\min} = k_B T \ln 2 \approx 2.87 \times 10^{-21} \text{ J} \quad \text{at } T = 300 \text{ K.}$$

If transition t erases b_t bits, then $E_{\text{total}} \geq k_B T \ln 2 \sum_t b_t$. Lloyd adds an upper limit on speed: "the speed with which a physical device can process information is limited by its energy" [T14, abstract], at most $2E/\pi\hbar$ operations per second for energy E , where $\hbar = 1.0546 \times 10^{-34}$ J s is the reduced Planck constant. Real chips spend many orders of magnitude above the Landauer floor per operation; the RCE project has no measured joules-per-operation figure at the time of writing, and H9 (Section 5 below) is the planned measurement. Kolmogorov flagged the missing cost dimension in 1965: his complexity measures "do not allow for the 'difficulty' of preparing a program p for passing from an object x to an object y " [T28, Sec. 4, p. 167]. Wolpert, quoting Boyd, notes as a caution that "different implementations of the same computation (input to output) can have radically different thermodynamic consequences" [T15, Sec. I.B]: energy belongs to the program and the machine together.

Definition (kinematic program). P is kinematic if, for every s_0 and every finite event stream,

$$(i) \text{ Var}[s_T] = 0, \quad (ii) E_{\text{total}} = \sum_{t=0}^{T-1} c(s_t, e_t) \leq c_{\max} F$$

where $c : S \times E \rightarrow \mathbb{R}_{\geq 0}$ is a cost table in joules for a stated machine, $c_{\max}$ its largest entry, and F the fuel bound (maximum steps) from the termination proof. Condition (i) says the trajectory has zero variance; condition (ii) says its cost can be computed, or at least bounded, before it runs.

Contrast. A stochastic generator returns a sample $x \sim p_\theta(x \mid \text{prompt})$ from a model with parameters θ . Its output is a distribution, its correctness is a probability, and its cost is known only after sampling. RCE-JEPA keeps the generator stochastic but makes the thing it generates kinematic, and puts a deterministic checker between them.

Part 4. A world model whose physics is known exactly

Most world models are built in fog. A robot learns physics from blurry cameras; a game agent infers rules it cannot read. Code is the rare world where the rules are written down.

World model. In the JEPA sense [LeCun 2022, cited above], a world model is a predictor $\hat{z}_{t+1} = g_\phi(z_t, a_t)$ of the next latent representation $z_{t+1} = \text{enc}(s_{t+1})$ from the current one and an action a_t , with learned parameters ϕ . For code the world is the Form's state space and the action is an event, so the target is computed exactly by δ :

$$\mathcal{L}(\phi) = \mathbb{E} \|g_\phi(\text{enc}(s_t), e_t) - \text{enc}(\delta(s_t, e_t))\|^2.$$

There is no aleatoric (irreducible) noise in the target. MuZero starts from the observation that lookahead planning algorithms "all rely on knowledge of the environment's dynamics, such as the rules of the game or an accurate simulator", and learns a model instead [T1, Sec. 1]; for code, those rules are δ , available in full.

Intent decoding as an inverse problem. Given an intent ι stated as constraints, find a Form whose trajectories satisfy it. This is written as energy minimization in the sense of energy-based models [T36, T2]:

$$\mathcal{E}(\iota, f) = \alpha V(f) + \beta d(\tau(f), \sigma(\iota)) + \gamma C(f), \quad z^* = \arg \min_z \mathcal{E}(\iota, \text{dec}(z)).$$

Here $V(f)$ is the number of checker violations (dimensionless); $\tau(f)$ the set of trajectories of f on the intent's example events; $\sigma(\iota)$ the trajectories the intent requires; d a distance between them (dimensionless count of mismatched states); $C(f)$ the predicted energy per run in joules; $\alpha, \beta \geq 0$ dimensionless weights and $\gamma \geq 0$ in J^{-1} ; dec the decoder from latent point to Form. The minimizer is decoded, then checked; the checker, not $\mathcal{E}$, decides acceptance. Du and Mordatch observe that "summation of two or more energies corresponds to satisfying all their goals or constraints" [T36, Sec. 1], which is why the three terms can be added. LeCun et al. add a caution: "combining two, separately trained energy-based models is not straightforward: there is no a priori guarantee that their energy scales are commensurate" [T2, Sec. 1.3, p. 6]. The weights α, β, γ must therefore be calibrated, not assumed, and their values are reported with each experiment.

Part 5. Consequences as testable predictions

A thesis that cannot lose is not a thesis. Each part above maps to a test that could prove it wrong.

- Part 1 and T2 (compression into Form) are tested by H1, H3 and H10.
- Part 3 (kinematics, energy accounting) is tested by H2, H4, H8 and H9.
- Part 4 (world model, inverse problem) is tested by H5 to H7.
- The proposer is tested by H11 to H13. All thirteen hypotheses, with metrics and thresholds, are in one table in numerical order in Section 5 below.

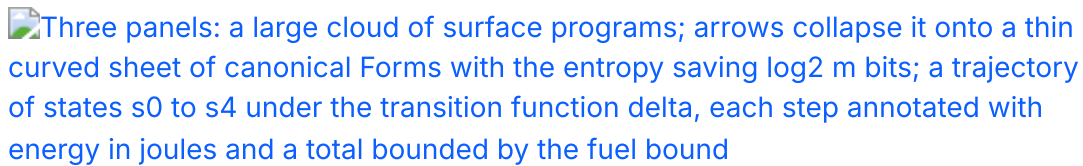
Three panels: a large cloud of surface programs; arrows collapse it onto a thin curved sheet of canonical Forms with the entropy saving $\log_2 m$ bits; a trajectory of states s_0 to s_4 under the transition function δ , each step annotated with energy in joules and a total bounded by the fuel bound

Figure 5. Information and kinematics. Left: program space, where many surface programs X express the same behaviour. Middle: canonicalization κ collapses each class onto one Form on a low-dimensional manifold Ω , saving $\Delta H = \log_2 m$ bits for a class of m equally likely variants. Right: a Form moves deterministically, $s(t+1) = \delta(s(t), e(t))$, each transition carrying a cost c in joules, with the total bounded by $c_{\text{max}} \cdot F$ before execution. A conceptual diagram, not data. [Open full size](#) · [PNG](#).

The provable chain: hypothesis, layer, evidence

A claim about "Laws of Coding" is only as strong as its weakest link. The table names, for each hypothesis, which of the four layers of law it tests, then follows it through three layers of implementation: the algorithm (the mathematics of the Form and the predictor), the software environment (checker, corpus, harness) and the hardware (the machine and its meter). Each link carries one of three labels. **[C] cited literature:** a published result, read as stated in the ledger. **[M] measured data:** a number measured by the OpenIE authors, with its source. **[P] planned experiment:** a test not yet run. At the time of writing no link carries data measured for this proposal. The figure below was drawn when three cells (H1, H7, H8) were coloured as measured; those cells are treated as planned until the numbers can be reproduced from public repositories. The chain is a plan for turning [P] into [M], and any [P] can still break it.

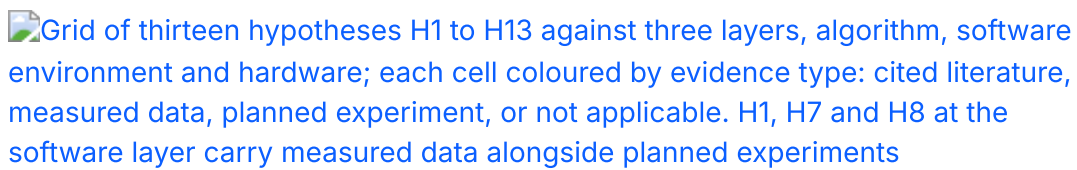
Grid of thirteen hypotheses H1 to H13 against three layers, algorithm, software environment and hardware; each cell coloured by evidence type: cited literature, measured data, planned experiment, or not applicable. H1, H7 and H8 at the software layer carry measured data alongside planned experiments

Figure 6. The provable chain for "Laws of Coding". Rows: hypotheses H1 to H13, each tagged with the layer of law it tests. Columns: implementation layers, algorithm, software environment, hardware. Blue: cited literature. Green: measured data. Orange: planned experiment. Grey: not applicable. The table below gives the content and source of every cell. [Open full size](#) · [PNG](#).

ID	Hypothesis	Layer of law tested	Algorithm	Software environment	Hardware
H1	Form much smaller than source	"Laws of Programming Languages" (canonical meaning) and "Laws of Coding"	[C] Chain rule: predicting the Form saves $H(X \mid \kappa(X))$ bits [T10]	[P] ρ on the canonical serialization over the lift corpus	Not applicable (a count of bits)
H2	Checker cheap relative to generation	"Laws of Programming Languages"	[C] Validating a result can be easier than proving the generator: CompCert's register allocator (Leroy, Section 4 below)	[P] Checker wall time and cycles per candidate	[P] $C_{\text{check}}/E_{\text{model}}$ in joules on a named meter
H3	Latent prediction reaches useful acceptance	"Laws of Coding"	[C] Prediction in representation space (I-JEPA, Assran et al., Section 3 below)	[P] pass@1 on held-out intent-to-Form tasks	Not applicable
H4	Energy per accepted program falls	"Laws of Computation"	[C] Landauer floor $k_B T \ln 2$ per erased bit [T11, T12]	[P] J_{acc} for both arms in one harness	[C] About 3 to 4 J per token for a 65B model (Samsi et al.); memory access dominates arithmetic [T26]. [P]

ID	Hypothesis	Layer of law tested	Algorithm	Software environment	Hardware
					Device meter, idle subtracted
H5	Energy ranking favours valid Forms	"Laws of Coding"	[C] Energies add to combine constraints [T36]	[P] AUROC of energy for accept against reject	Not applicable
H6	Reference apps are encodable	"Laws of Programming Languages"	[C] No general procedure decides non-trivial semantic properties [T9, T3], hence the restricted language	[P] 7 of 7 reference apps as checked Forms	[P] Replay of the same Forms across machines (with H8)
H7	Security independent of the model	"Laws of Programming Languages"	[C] After about six CPU-years, Csmith found no wrong-code errors in CompCert (Yang et al., Section 4 below)	[P] Fuzzed and model-generated adversarial Forms	[C] Speculative-execution side channels lie outside an instruction set's semantics [T26]; out of the Form's scope, stated
H8	Determinism	"Laws of Programming Languages" and "Laws of Computation"	[C] Any deterministic computation can be made logically reversible given a copy of its input [T12]	[P] Bit-identical replay of 10,000 event streams	[C] Interrupts arrive "at unpredictable and irreproducible moments" [T8]. [P] 3 machines and operating systems
H9	Cost predicted before execution	"Laws of Computation"	[C] Energy bounds speed [T14]; cost bound $c_{\max} F$ (Part 3 above)	[P] Cost-table prediction per Form	[C] Implementations of one computation differ in thermodynamic cost [T15]. [P] Median error within 20% of measured joules
H10	Learning on the quotient saves data	"Laws of Coding"	[C] Better prediction is better compression [T32]; minimum description length [T33]	[P] Equal acceptance with at most half the training examples	[P] Training energy measured for both predictors
H11	Energy monotone in edit distance	"Laws of Coding"	[C] A monotone cost ranking makes plans that deviate further from the goal cost more (Chahe et al., [B19] in the proposer section)	[P] Share of families where energy rises with edit distance	Not applicable
H12	Decision head calibrated on gate outcomes	"Laws of Coding" and "Laws of Computation"	[C] Calibrated typed decisions are measurable: pooled Choice ECE 0.028 for	[P] ECE per decision type after 10^4 receipts, against	[P] Latency and joules per decision on a named device

ID	Hypothesis	Layer of law tested	Algorithm	Software environment	Hardware
			one hosted decision model ([B7])	a frequency table	
H13	Discrete against continuous latent	"Laws of Coding"	[C] Discrete tokens kept long-horizon prediction stable where continuous latents accumulated error ([B3])	[P] Planning success at edit lengths 1, 5 and 20	Not applicable

Who holds these views and why

Ideas rarely arrive alone. Each part of the thesis echoes arguments made, for their own reasons, by people who have never heard of RCE. They are listed as convergence observations: agreement in direction, never confirmation of the RCE project's claims. Where the OpenIE authors read a source in full it is quoted with its section; where they did not, the entry says so and quotes nothing.

Those who already think this way. The idea that programming and computation should have laws of their own did not originate with the RCE project, and it is older than most of the field. This page joins a conversation that has run for at least forty years and lists it as convergence, not priority.

- **Hoare and colleagues, "Laws of Programming" (1987).** Nine authors argued that "conventional procedural programs are mathematical expressions, and that they are subject to a set of laws as rich and elegant as those of any other branch of mathematics, engineering, or natural science" [T7, p. 672], and gave nearly one hundred of them. They also named the gap that remains (quoted in the timeline above) [T7, p. 686].
- **Bennett and Landauer, "The Fundamental Physical Limits of Computation" (1985).** "We are looking for general laws that must govern all information processing, no matter how it is accomplished" [T13, p. 48]. They add a caution this page shares: "To the extent that we have found limits, they are terribly far away from the real limits of modern technology" [T13, p. 48]. Their result for reversible machines: "There is thus no minimum amount of energy that must be expended in order to perform any given computation" [T13, p. 51], while erasure and error correction still dissipate, since, in their words, "Any correctional device must therefore dissipate some energy" [T13, p. 51].
- **McKee, "A few (proposed) fundamental laws of computation" (1995).** A short essay in IEEE Computer proposing laws for the field [T31]. The OpenIE authors have not obtained the text, so it is not quoted or summarized beyond its title.
- **Carette, James and Sabry, "Embracing the laws of physics" (2022).** They note that quantum theory and thermodynamics "imply that information is a conserved quantity of physical processes and hence of primitive computational operations", and set out "to redevelop foundational computational models in a way that embraces the principle of conservation of information" [T16, abstract, PDF p. 2]. Their closing summary: "The

entire edifice of computer science including its mainstream models of computations, programming languages, and logics is founded on classical physics" [T16, Sec. 5.5, p. 45].

- **Lehman (1980)** states five empirical regularities of software evolution for programs embedded in the world [T6]. The first, "Continuing Change": a program "that is used and that as an implementation of its specification reflects some other reality, undergoes continual change or becomes progressively less useful" [T6, Table I, p. 1068]. He is explicit that these are not physics: "They are, however, clearly not immutable" in the way physical and chemical regularities are, since they "arise from the habits and practices of people and organizations" [T6, Sec. V-A, p. 1069], the same distance this page keeps between "Laws of Coding" and "Laws of Computation". **Brooks** distinguished "essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity" from accidental ones [T23, PDF p. 1]. **Dijkstra** observed that a computer spans a ratio of time grains of " 10^{10} or more", which makes it "the first to provide us with an environment where highly hierarchical artefacts are both possible and necessary" [T8].

The groups below each hold one strand of the same idea.

Compression as understanding (Part 1, T2). Solomonoff, Kolmogorov and Chaitin defined complexity as shortest description [T27, T28, T29]; Rissanen made it a model selection rule [T33]; Hutter's prize and AIXI and Sutskever's talks extend it to intelligence (not read here; not quoted). Delétang and colleagues at DeepMind argue that "a model that compresses well generalizes well" [T32, Sec. 1] and show large language models compress images and audio. Schmidhuber argues that if the history of the universe were computable, its simplest explanation "would be the shortest program that computes it" [T34, Sec. 1]. Disagreement: Delétang et al. warn that "scaling beyond a certain point will deteriorate the compression performance since the model parameters need to be accounted for" [T32, Sec. 1], and Chollet argues that "solely measuring skill at any given task falls short of measuring intelligence" [T37, abstract]; compression of a fixed corpus is not the same as efficient learning of new skills.

Computation as physics (Part 3). Landauer [T11], Bennett [T12, T30] and Lloyd [T14] hold that information processing is a physical process with energy costs and limits. Lloyd: "Energy limits speed. Entropy limits memory." [T14, Sec. 1.5]. Wolpert's stochastic thermodynamics treats computation, including finite automata, as non-equilibrium physics: "One of the most prominent aspects of computers is that they are inherently nonequilibrium systems" [T15, abstract]; finite automata are treated in Sec. XII [T15]. Fredkin and Toffoli's conservative logic [T38], Feynman's lectures on computation, Wheeler's "it from bit" and Wolfram's computational universe point the same way (not read here). Disagreement: Bennett records objections that Landauer's principle is false, or, being a consequence of the second law of thermodynamics, either unnecessary or insufficient to exorcise Maxwell's demon, and answers them [T12]; the debate is about foundations, not about the size of the floor. A second disagreement is historical: Landauer wrote "We believe that devices exhibiting logical irreversibility are essential to computing" [T11, Sec. 3, p. 186], and Bennett's 1973 construction

answered that they are not, at the price of about twice as many steps and a large temporary history [T30, pp. 525, 529]; Bennett then estimated " $kT \ln 2$ is, as Landauer has argued [1], an approximate lower bound" only for machines that throw away about one bit per step [T30, p. 532].

State machines and deterministic semantics (Part 1, T3). Turing's machines [T9], Milner's calculus [T18], Harel's statecharts [T19], Lamport's TLA, which treats a program as a set of behaviours over states [T20], the synchronous languages Esterel and Lustre (Berry, Halbwachs) [T21, T22], Elliott's functional reactive programming [T39], the Elm architecture, deterministic simulation testing at FoundationDB and TigerBeetle, and Hoare's and Dijkstra's view of programs as predicates [T17] all treat a program as a state machine with exact semantics. They are cited here as history. The FoundationDB and TigerBeetle pages and the Elm guide were read in full for the convergence section below (its FoundationDB, TigerBeetle and Elm Architecture rows); the others were not read in full for this section, so none of them are quoted. Limits: Turing [T9] and Rice [T3] bound what can be decided about arbitrary programs; deterministic semantics is easy to state for a closed language and hard to keep across real hardware, threads and networks.

Programs as data on a low-dimensional manifold (Part 2). Hindle et al. found software "natural", more repetitive than English [T24] (not read in full here). Alon et al.'s code2vec maps code to vectors where "semantically similar objects are mapped to close vectors" [T35, Sec. 1]. Sketch-based synthesis (Solar-Lezama), DreamCoder's library learning (Ellis et al. 2021, cited in Section 2 below; its supplement describes library learning as "automatic refactoring" [Ellis et al. 2021, supplement S3.2, p. 39]) and Allamanis's survey of machine learning on code point the same way (Solar-Lezama and Allamanis not read here). Disagreement: code2vec's authors report that "When given uninformative, obfuscated or adversarial variable names, the prediction of the label is usually less accurate" [T35, Sec. 7]: learned embeddings can track names rather than meaning, which is the gap a canonical Form is meant to close.

World models and latent prediction (Part 4). LeCun's JEPA programme at Meta FAIR (cited above), Ha and Schmidhuber, who "learn a compressed spatial and temporal representation of the environment" [T40, abstract], MuZero [T1], physics-informed neural networks, where prior physical knowledge "can act as a regularization agent that constrains the space of admissible solutions to a manageable size" [T41, Sec. 1], and Hamiltonian neural networks, motivated by models whose simulation "drifts over time to higher or lower energy states" [T42, Sec. 1], all build structure into learned dynamics. Lagrangian networks (Cranmer), Dreamer (Hafner) and Koopman operator learning share the aim (not read here). Disagreement: Ha and Schmidhuber found that a learned world model "will be exploitable by the controller, even if in the actual environment such exploits do not exist" [T40, Sec. 4.5]; Greydanus et al. note their method assumes a conserved quantity and so cannot model friction [T42, Sec. 3.2]. For code, the checker is the guard against exploiting the model's errors.

Energy-based models (Part 4). Hopfield [T43] and Hinton [T44] framed learning as shaping an energy surface (not read here); Du and Mordatch [T36] show implicit generation by descending a learned energy, and LeCun (cited above) places energy minimization at the

centre of JEPA inference. Disagreement within the field concerns training stability and sampling cost, which Du and Mordatch discuss [T36].

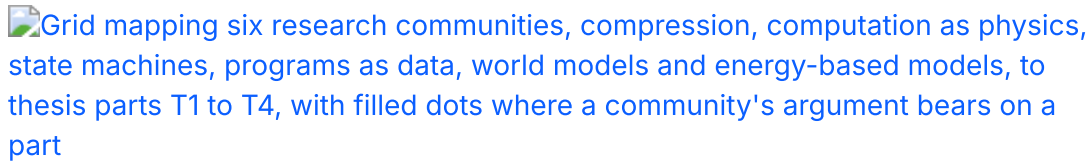


Figure 7. Which communities bear on which part of the thesis. A filled dot marks a convergence observation, not confirmation. Part 1: finite information system; T2: compression into Form; T3: kinematics and energy; T4: world model and inverse problem. [Open full size](#) · [PNG](#).

Thesis references

Ledger: Full = read in full; Partial = read in part; Unavailable = not read (paywalled or not retrieved), listed with identifier only and never quoted.

1. [T1] J. Schrittwieser et al., "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model", Nature 588 (2020). [arXiv:1911.08265](#). Full.
2. [T2] Y. LeCun et al., "A tutorial on energy-based learning", 2006. [PDF](#). Full.
3. [T3] H. G. Rice, "Classes of recursively enumerable sets and their decision problems", Trans. AMS 74 (1953). [doi:10.1090/S0002-9947-1953-0053041-6](#). Unavailable.
4. [T4] M. A. Di Liscia, "Johannes Kepler", Stanford Encyclopedia of Philosophy. [plato.stanford.edu](#). Full.
5. [T5] G. E. Smith, "Newton's Philosophiae Naturalis Principia Mathematica", Stanford Encyclopedia of Philosophy. [plato.stanford.edu](#). Full.
6. [T6] M. M. Lehman, "Programs, life cycles, and laws of software evolution", Proceedings of the IEEE 68(9) (1980). [doi:10.1109/PROC.1980.11805](#). Full.
7. [T7] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey and B. A. Sufrin, "Laws of Programming", Communications of the ACM 30(8), 672-686 (1987). [doi:10.1145/27651.27653](#); [PDF copy](#). Full.
8. [T8] E. W. Dijkstra, "The Humble Programmer", ACM Turing Lecture 1972, EWD 340. [Transcription](#); [doi:10.1145/355604.361591](#). Full.
9. [T9] A. M. Turing, "On computable numbers", Proc. London Math. Soc. 42 (1936). [doi:10.1112/plms/s2-42.1.230](#). Unavailable.
10. [T10] C. E. Shannon, "A mathematical theory of communication", Bell System Technical Journal 27 (1948). [doi:10.1002/j.1538-7305.1948.tb01338.x](#); [reprint](#) (page numbers refer to this reprint). Full (all 55 pp. of the reprint, Parts I to V and Appendices 1 to 7).
11. [T11] R. Landauer, "Irreversibility and heat generation in the computing process", IBM J. Res. Dev. 5 (1961). [doi:10.1147/rd.53.0183](#). Full.
12. [T12] C. H. Bennett, "Notes on Landauer's principle, reversible computation, and Maxwell's Demon", 2003. [arXiv:physics/0210005](#). Full.
13. [T13] C. H. Bennett and R. Landauer, "The Fundamental Physical Limits of Computation", Scientific American 253(1), 48-57 (1985). [doi:10.1038/scientificamerican0785-48](#); [PDF copy](#). Full (all text pages of the linked copy, pp. 48-56).

14. [T14] S. Lloyd, "Ultimate physical limits to computation", Nature 406 (2000). [arXiv:quant-ph/9908043](#). Full.
15. [T15] D. H. Wolpert, "The stochastic thermodynamics of computation", J. Phys. A 52 (2019). [arXiv:1905.05669](#). Full.
16. [T16] J. Carette, R. P. James and A. Sabry, "Embracing the laws of physics: Three reversible models of computation", Advances in Computers (2022). [doi:10.1016/bs.adcom.2021.11.009](#); [author copy](#). Full.
17. [T17] C. A. R. Hoare, "An axiomatic basis for computer programming", CACM 12 (1969). [doi:10.1145/363235.363259](#). Unavailable.
18. [T18] R. Milner, A Calculus of Communicating Systems, LNCS 92 (1980). [doi:10.1007/3-540-10235-3](#). Unavailable.
19. [T19] D. Harel, "Statecharts: a visual formalism for complex systems", Sci. Comput. Program. 8 (1987). [doi:10.1016/0167-6423\(87\)90035-9](#). Unavailable.
20. [T20] L. Lamport, "The temporal logic of actions", ACM TOPLAS 16 (1994). [doi:10.1145/177492.177726](#). Unavailable.
21. [T21] G. Berry and G. Gonthier, "The Esterel synchronous programming language", Sci. Comput. Program. 19 (1992). [doi:10.1016/0167-6423\(92\)90005-V](#). Unavailable.
22. [T22] N. Halbwachs et al., "The synchronous data flow programming language LUSTRE", Proc. IEEE 79 (1991). [doi:10.1109/5.97300](#). Unavailable.
23. [T23] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", IEEE Computer 20(4) (1987; first presented 1986). [doi:10.1109/MC.1987.1663532](#); [PDF copy](#). Full.
24. [T24] A. Hindle et al., "On the naturalness of software", ICSE (International Conference on Software Engineering) 2012. [doi:10.1109/ICSE.2012.6227135](#). Unavailable.
25. [T25] J. W. Backus et al., "The FORTRAN Automatic Coding System", Western Joint Computer Conference (1957). [Computer History Museum copy](#). Full.
26. [T26] J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture", Communications of the ACM 62(2) (2019). [doi:10.1145/3282307](#); [CACM web text](#). Full.
27. [T27] R. J. Solomonoff, "A formal theory of inductive inference, Part I", Information and Control 7 (1964). [doi:10.1016/S0019-9958\(64\)90223-2](#). Unavailable.
28. [T28] A. N. Kolmogorov, "Three approaches to the quantitative definition of information", Problems of Information Transmission 1 (1965); English reprint, International Journal of Computer Mathematics 2 (1968), [doi:10.1080/00207166808803030](#). Full (page numbers from the 1968 reprint).
29. [T29] G. J. Chaitin, "On the length of programs for computing finite binary sequences", J. ACM 13 (1966). [doi:10.1145/321356.321363](#). Unavailable.
30. [T30] C. H. Bennett, "Logical Reversibility of Computation", IBM J. Res. Dev. 17(6), 525-532 (1973). [doi:10.1147/rd.176.0525](#). Full.
31. [T31] G. McKee, "A Few (Proposed) Fundamental Laws of Computation", IEEE Computer 28(1), 120 (1995), Open Channel. No DOI located; [DBLP table of contents](#). Unavailable.
32. [T32] G. Delétang et al., "Language Modeling Is Compression", ICLR 2024. [arXiv:2309.10668](#). Full.

33. [T33] J. Rissanen, "Modeling by shortest data description", *Automatica* 14 (1978). [doi:10.1016/0005-1098\(78\)90005-5](#). Unavailable.
34. [T34] J. Schmidhuber, "Driven by Compression Progress", 2009. [arXiv:0812.4360](#). Full.
35. [T35] U. Alon et al., "code2vec: Learning Distributed Representations of Code", *POPL* 2019. [arXiv:1803.09473](#). Full.
36. [T36] Y. Du and I. Mordatch, "Implicit Generation and Modeling with Energy-Based Models", *NeurIPS* 2019. [arXiv:1903.08689](#). Full.
37. [T37] F. Chollet, "On the Measure of Intelligence", 2019. [arXiv:1911.01547](#). Full.
38. [T38] E. Fredkin and T. Toffoli, "Conservative logic", *Int. J. Theor. Phys.* 21 (1982). [doi:10.1007/BF01857727](#). Unavailable.
39. [T39] C. Elliott and P. Hudak, "Functional reactive animation", *ICFP* 1997. [doi:10.1145/258948.258973](#). Unavailable.
40. [T40] D. Ha and J. Schmidhuber, "World Models", 2018. [arXiv:1803.10122](#). Full.
41. [T41] M. Raissi, P. Perdikaris and G. E. Karniadakis, "Physics Informed Deep Learning (Part I)", 2017. [arXiv:1711.10561](#). Full.
42. [T42] S. Greydanus, M. Dzamba and J. Yosinski, "Hamiltonian Neural Networks", *NeurIPS* 2019. [arXiv:1906.01563](#). Full.
43. [T43] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities", *PNAS* 79 (1982). [doi:10.1073/pnas.79.8.2554](#). Unavailable.
44. [T44] G. E. Hinton, "Training products of experts by minimizing contrastive divergence", *Neural Computation* 14 (2002). [doi:10.1162/089976602760128018](#). Unavailable.

Convergence by commitment

The RCE thesis rests on seven commitments. Each has been reached independently, often more than once, by people working for their own reasons. The tables below list projects whose primary sources (paper, official documentation or repository) were opened and checked against the sentence written here. Each row says what the project established and where it stops relative to the commitment. These are convergence observations: agreement in direction, not confirmation of RCE's claims and not a ranking. Items from OpenIE, the organization behind RCE, are marked *same house* and are never used as evidence.

Acronyms used in this section: RCE, the Reality Code Engine; AST, abstract syntax tree; IR, intermediate representation; LLM, large language model; JSON, JavaScript Object Notation; WASM, WebAssembly; SMT, satisfiability modulo theories (Z3 is an SMT solver); DST, deterministic simulation testing; CEGIS, counterexample-guided inductive synthesis.

 [Grid of seven commitments, canonical form, log and replay, projection, stored intent, deterministic gate, promotion and reuse, and energy or cost, against verified outside projects; a filled dot marks a commitment that the project's own primary source states](#)

Figure 8. Which commitment each verified outside project reaches, according to its own primary source. A dot is a convergence observation, not a score; RCE is not drawn. Select the figure to enlarge it. [Open full size](#) · [PNG](#).

1. One canonical form; surface text is a view

Project	What it established	Where it stops
Intentional Programming, Simonyi, 1995 [C1]	A program is a tree of nodes, each an instance of an intention; users edit the tree and view it in any notation; computational intent can be kept separate from implementation	A research system at Microsoft; the report describes no determinism or replay model (a transaction file feeds version control, but no replay semantics are given)
Unison [C2]	Each definition is identified by a hash of its syntax tree; names are separately stored metadata	The hash is of structure: two definitions with the same behaviour and different structure have different hashes
Equality saturation and egg [C3; C4]	An e-graph holds many equivalent versions of a program at once; a cost model picks one (Tate et al.: a pseudo-Boolean solver over a static cost model; egg: an extractor with a user-provided cost function)	An optimization and synthesis technique applied per function (intraprocedural in Tate et al.) or over terms in a user-defined language (egg, which has no built-in support for binding); saturation is bounded when it does not terminate, and the e-graph is not kept as the program's stored form
Tacit, 2026 [C5]	The AST is the source of truth; every valid AST has one byte-exact canonical text; definitions are addressed by BLAKE3 hash; malformed code becomes typed Hole nodes	One target (LLVM); no state-machine semantics stated
X07, 2026 [C6]	Canonical source is a JSON AST, edited with JSON Patch; a 646-code diagnostic catalogue with enforced quickfix coverage	Canonical bytes, not canonical behaviour
Codifide, 2026 [C7]	The canonical form is a typed hypergraph with a deterministic byte form and SHA-256 hash; surface syntax is a projection; intent is part of a definition's identity	Described by its authors as a contract-and-dispatch language, not general computation
NAIL [C8]	nail canonicalize normalizes any program; nail check --strict rejects non-canonical input	An RFC 8785-inspired JSON subset that, in its own words, does not claim full RFC 8785 compliance; byte canonicalization only
EqNet, Allamanis et al., 2017 [C9]	Neural representations trained on known equivalence classes so that equivalent expressions get identical or highly similar vectors	Boolean and polynomial expressions (supervised), not programs

2. A program is a deterministic state machine over a log

Project	What it established	Where it stops
Abstract State Machines, Gurevich, 2000 [C10]	The paper's title states the result: sequential abstract state machines capture sequential algorithms	A theorem, not a system: every sequential algorithm, defined by three postulates (sequential time, abstract state, bounded exploration; Def. 6.1), has an equivalent sequential abstract state machine that matches it step for step (Main Theorem 6.13). That the postulates cover all sequential algorithms is argued, not proved
Event sourcing, Fowler, 2005 [C11]	"Capture all changes to an application state as a sequence of events"	A pattern, published as a draft; replay stays consistent only if gateways record external query responses and suppress external updates during replay
The Elm Architecture [C12]	Model, View, Update: state, a view of the state, and an update from messages	One language and one target (the browser)
Dedalus and the CALM theorem, Hellerstein, Alvaro et al. [C13; C14]	Dedalus: Datalog with explicit time, modelling mutable state and asynchronous messaging. CALM: a program has a consistent, coordination-free distributed implementation if and only if it is monotonic (proved by Ameloot et al.; [C14] gives a proof sketch)	Dedalus and Bloom are research languages and CALM is a theorem; neither emits to other languages
Durable execution: Temporal, Restate [C15; C16]	Temporal: workflow code "must be deterministic to support replay". Restate: every step and its result are recorded in a journal	Replay serves resilience, not program identity
TigerBeetle deterministic simulation testing, 2026 [C17]	Logical determinism (like FoundationDB) plus physical determinism: replicas converge to byte-by-byte identical state; real code runs in a simulator	One database, built by hand to this discipline
FoundationDB simulation testing [C18]	"A deterministic simulation of an entire FoundationDB cluster within a single-threaded process"; determinism "allows perfect repeatability of a simulated run"	A testing discipline for one database, not a program form
musil [C19]	An explicit-state model checker as a Python library; its simulate mode runs the real node code on a seeded, replayable simulated network	Bounded state spaces; it checks the model, and simulation finds bugs rather than proving their absence

Project	What it established	Where it stops
AILANG; agent-scroll [C20; C21]	AILANG: every construct has deterministic semantics, with replayable execution and structured traces. agent-scroll: canonical, byte-deterministic, hash-chained transcripts with a verifier that catches any single-byte change	AILANG is functional, not a state-machine form; agent-scroll logs conversations, not programs

3. Every target is a projection: write once, emit many

Project	What it established	Where it stops
MLIR [C22]	A reusable compiler infrastructure spanning levels of abstraction, hardware targets and domains	Starts below the source: it has no parser generation or AST construction (Sec. 7), and its paper names translation validation across an extensible dialect ecosystem as an open problem (Sec. 2)
Model Driven Architecture, OMG, 2003 [C23]	Computation-independent, platform-independent and platform-specific viewpoints	Petre reports, from Forward and Lethbridge, that keeping the model up to date with the code was the biggest perceived problem (68% agreement) [C24]
Protocol Buffers; Lumos [C25; C26]	Define a schema once and generate bindings in many languages (Lumos: Rust and TypeScript for Solana)	Data shapes, not behaviour; the same Protocol Buffers message can have many binary serializations ("You cannot compare two messages for equality without fully parsing them")
Forge, 2026 [C27]	"Many frontends feed one verified intermediate representation, and that representation feeds many backends"	Board and peripheral descriptions, not general programs
Faber Romanus, 2026 [C28]	One high-level IR projected to Rust, Go, TypeScript, Swift and Faber, and through a lower IR to LLVM and WASM text, with Metal and CUDA kernels; support is measured and stated per target	Agreement across targets is measured by corpus coverage, not proved
Intent, 2026 [C29]	One source file compiles to Rust, JavaScript and WASM; contracts are enforced at runtime in every target and optionally checked by Z3	Three targets
NanoLang [C30]	Core semantics proved in Coq with no axioms; C is the production target	PTX and RISC-V are experimental; LLVM and WASM are planned

4. Store intent; derive the implementation

Project	What it established	Where it stops
FORTTRAN, Backus et al., 1957; deductive synthesis, Manna and Waldinger, 1980 [C31; C32]	Automatic coding from formulas; programs extracted from constructive proofs	FORTTRAN became a compiler; deductive synthesis had no implementation at the time
Spec-driven development, as classified by Böckeler [C33]	Three levels: spec-first, spec-anchored and spec-as-source, in which "only the spec is edited by the human, the human never touches the code"; of Kiro, spec-kit and Tessl, only Tessl explores spec-as-source	The spec is natural language, and Böckeler saw code generated repeatedly from one spec differ; nothing checks that the code is the spec
Vericoding benchmark, 2025 [C34]	12,504 formal specifications in Dafny, Verus and Lean; at least one of nine off-the-shelf LLMs, with five attempts each, solved 82% (Dafny), 44% (Verus) and 27% (Lean) of the tasks, and the best single model reached 67.5%, 30.9% and 17.9% (Table 3); adding natural-language descriptions gave no statistically significant gain on the one subset tested (Verina, 157 tasks; Sec. 4.1)	Specification languages are the proof languages; no runtime form or multi-target emission; among successes about 9% of specs were too weak to capture intent (Sec. 4)
Codifide [C7]	Every definition carries an intent that compilation never discards	Contract-and-dispatch scope

Recommendation. The vericoding benchmark [C34] is used as an outside benchmark for the step from specification to Form: a slice of its state-machine-shaped Dafny, Verus and Lean specifications is lifted into Forms and the share the RCE gate accepts is reported beside the published vericoding results.

5. The model proposes; a deterministic gate decides

Project	What it established	Where it stops
CompCert; seL4 [C35; C36]	Machine-checked compiler and kernel; CompCert checks an unverified heuristic's output with a verified checker; seL4's proof cost about 20 person-years	Proof effort per artifact
Sketching, Solar-Lezama et al., 2006 (the counterexample-driven loop later named CEGIS) [C37]	Against a human-written partial program (sketch) and an executable specification, a synthesizer proposes, a checker returns a counterexample, and the loop repeats	Proposers are SAT solvers, not learned models; limited to finite programs (bounded input, guaranteed termination), with guarantees only for the bounded instance

Project	What it established	Where it stops
AlphaProof; Clover; AutoVerus [C38; C39; C40]	A model proposes; Lean, Dafny or Verus checks	Gate on proofs or code text; no canonical program form
Agentic model checking, 2026; LLM-driven verified synthesis for concept design, 2026 [C41; C42]	"Agents propose, solvers verify" (an LLM still triages counterexamples for realism); in the second, invariants alone under-constrained designs, which differed across runs, some of them implausible	[C41] verifies existing code with LLM-inferred specs; checks are bounded (default unwind 4) and spec-relative, and the authors name over-weak specs that "cause bugs to be missed silently" as the primary failure mode. [C42] synthesizes coordination rules for design models (not code), checked only within a bounded Alloy scope, with an LLM translating the design into Alloy. Both warn that validity is not intent
Vera, 2026; Vow; Thermite [C43; C44; C45]	Vera: mandatory contracts, proved by the Z3 SMT solver where decidable and otherwise checked at run time: "The model doesn't need to be right, it needs to be checkable". Vow: contracts statically verified by ESBMC bounded model checking. Thermite: contracts with counterexamples on failure and a recorded assurance level (Verus/Z3, Lean, Kani bounded, runtime)	Gate at the function; no replay or promotion
Decision models, a class of concept [C46]	A model whose entire output is a typed answer (one option from a closed set, a bounded score, or a yes/no probability) with a probability for every option, trained for calibration	Needs a closed option set. The RCE project builds its own decision head and uses no outside decision model

6. Accepted work is promoted and reused

Project	What it established	Where it stops
Gödel machine, Schmidhuber; Löbian obstacle, Yudkowsky and Herreshoff [C47; C48]	Self-rewrite only after a proof, under its own axioms, that the rewrite has higher utility than continuing to search; in the straightforward formalism, a proof-based agent cannot trust a successor that uses its own proof system (Löb's theorem), so trust shrinks with each rewrite, and the paper's technical workarounds are only partial	Theory

Project	What it established	Where it stops
Voyager; STOP; Darwin Gödel Machine; AlphaEvolve [C49; C50; C51; C52]	A growing library of executable skills; scaffolds that improve themselves (the model's weights are unchanged); archives of self-modified agents (kept if they still compile and can edit code; parents chosen by benchmark score); code evolved under automatic evaluators	Selection by score or evaluator, not by proof at an interface. Voyager's evaluator is a GPT-4 critic; AlphaEvolve's scores may include LLM-graded feedback, and its correctness came from construction, randomized testing or human and verification review outside the loop
Getafix; Sorald; TBar [C53; C54; C55]	Fix patterns mined from past fixes or written as templates, reused on new cases without a model; with perfect fault localization TBar correctly fixes 74 of 395 Defects4J bugs (plausible patches for 101), and 43 with standard fault localization	Patterns are per bug class, learned or written for one language (Getafix's method itself is language-agnostic); Getafix checks only its top fix, by re-running the static analyzer

7. Energy or cost is the objective

Project	What it established	Where it stops
Landauer, 1961; Bennett, 1973 [C56; C57]	Erasing a bit has a thermodynamic floor; computation can in principle be made logically reversible	Physics, not a software objective
FrugalGPT, 2023 [C58]	A cascade of models matched the best single model with up to 98% lower cost on its tasks	Dollar cost; no deterministic tiers below the models
Cost-of-pass, 2025; AI Agents That Matter, 2024 [C59; C60]	Expected monetary cost of a correct solution; report and jointly optimize cost with accuracy	Dollars and tokens, not joules; the Cost-of-Pass authors note energy could replace dollars but do not measure it
Compiler world models, 2026 [C61]	Rolls a tensor program's state forward through scheduling actions in latent space, then ranks candidate schedules within a workload before measuring on hardware (a relative score, not a latency prediction)	Tensor schedules only (TVM AutoScheduler); ranks candidates, does not generate them
Codifide [C7]	Selects among satisfied candidates by declared cost	Declared cost, not measured energy
JouleOS (OpenIE) [same house]	Energy budgets and the est_j, reported_j, measured_j labels	Same house: listed for disclosure, not as evidence

What the RCE project measured (OpenIE's own data)

The following are OpenIE's own results from [the RCE paper](#) (version 1.4), checked against the engine repository. They are reported here as the project's data, not as a comparison with the projects above.

- **Round trip.** On 10,000 preregistered generated Forms with a frozen engine and no model calls, Form IR to canonical JSON and back was the identity for 10,000 of 10,000. Two compiles gave the same bytes, and the XState and SCXML (State Chart XML) projections of the original and round-tripped Form were byte identical, in 10,000 of 10,000. The emitted Python printed the reference trace in 9,999 of 10,000.
- **Five targets.** The Rust, WASM, Python, XState and SCXML targets print the reference trace byte for byte on 529 events (29 edge events and 500 seeded random values) of one eleven-operator specification. This is differential testing on that input set, not a proof and not a 10,000-Form result.
- **Canonical form.** The canonical serialization is injective on Forms, so two Forms that differ have different shape hashes. Equivalence of behaviour is defined separately (fit equivalence), and equivalent variants are grouped by sound rewrites in an e-graph; the paper does not claim that equivalent behaviour always yields one canonical form.
- **Proofs.** Determinism, totality, replay as a fold of the log, and atomicity of refused events are proved in Lean 4 on a core calculus. The Rust program emitter is proved on a core as a verified translation into MiniRust, a deep embedding of the emitted Rust subset; that the Rust compiler implements MiniRust is trusted and tested. This proof covers the program emitter, not the Form projection.
- **Repair ladder, with the location given and token identity as the gate, zero model calls.** 520,541 of 520,821 one-token mutants (99.95%) in nine repositories; 13,586 of 21,320 single-statement Java bugs (63.7%) when the whole file and project indexes are supplied, and 9,288 of 24,072 (38.6%) from the seven-line hunk alone.
- **Gate.** Replay of recorded inputs, a shadow run with a coverage bound, an output schema check and a Kani proof certificate; a proposed rewrite also needs a Z3 proof. A local 1.5-billion-parameter model proposed 16 equivalences, of which the gate admitted 3.
- **Energy.** Every receipt carries `est_j`; an unpriced model call is never read as zero. `measured_j` (wall meter) is empty in every result so far.

Sources for this section

Labels as in the ledger: Full = the whole primary source (paper, documentation page, README or project description) was read, including references and appendices where present. Every source listed here was read in full on or before 14 October 2026; [C46] points to sources listed in the decision-head section.

1. [C1] C. Simonyi, "The Death of Computer Languages, the Birth of Intentional Programming", Microsoft Research MSR-TR-95-52, September 1995. [Microsoft Research](#); [PDF copy](#). Full.

2. [C2] Unison, "The big idea". unison-lang.org. Full.
3. [C3] R. Tate, M. Stepp, Z. Tatlock and S. Lerner, "Equality Saturation: A New Approach to Optimization", Logical Methods in Computer Science 7(1:10), 2011 (extended version of POPL 2009). [arXiv:1012.1802](https://arxiv.org/abs/1012.1802). Full.
4. [C4] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock and P. Panchekha, "egg: Fast and Extensible Equality Saturation", POPL 2021. [arXiv:2004.03082](https://arxiv.org/abs/2004.03082). Full.
5. [C5] Tacit. github.com/weetster/tacit. Full (README).
6. [C6] X07. github.com/x07lang/x07. Full (README).
7. [C7] Codifide. github.com/codifide/codifide-programming-language. Full (README).
8. [C8] NAIL. pypi.org/project/nail-lang. Full (project description).
9. [C9] M. Allamanis, P. Chanthirasegaran, P. Kohli and C. Sutton, "Learning Continuous Semantic Representations of Symbolic Expressions", ICML 2017 (PMLR 70). [arXiv:1611.01423](https://arxiv.org/abs/1611.01423). Full.
10. [C10] Y. Gurevich, "Sequential Abstract State Machines Capture Sequential Algorithms", ACM TOCL 1(1), 2000. [doi:10.1145/343369.343384](https://doi.org/10.1145/343369.343384); [PDF copy](#). Full.
11. [C11] M. Fowler, "Event Sourcing", 2005. martinfowler.com. Full.
12. [C12] The Elm Architecture. guide.elm-lang.org. Full.
13. [C13] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier and R. Sears, "Dedalus: Datalog in Time and Space", UC Berkeley EECS-2009-173, December 2009. [PDF](#). Full.
14. [C14] J. M. Hellerstein and P. Alvaro, "Keeping CALM: When Distributed Consistency is Easy", 2019. [arXiv:1901.01930](https://arxiv.org/abs/1901.01930). Full. The formal proof is T. J. Ameloot, F. Neven and J. Van den Bussche, "Relational transducers for declarative networking", JACM 60(2), 2013.
15. [C15] Temporal, "Workflow Definition". docs.temporal.io. Full.
16. [C16] Restate, "Key Concepts: Durable Execution". docs.restate.dev. Full.
17. [C17] C. Bhandari, "Protocol-Aware Deterministic Simulation Testing", TigerBeetle, 20 August 2026. tigerbeetle.com. Full.
18. [C18] FoundationDB, "Simulation and Testing". apple.github.io/foundationdb. Full.
19. [C19] musil (project description, version 0.35.6). pypi.org/project/musil. Full.
20. [C20] AILANG. github.com/sunholo-data/ailang. Full (README).
21. [C21] agent-scroll. github.com/p-vbordei/agent-scroll. Full (README).
22. [C22] C. Lattner et al., "MLIR: A Compiler Infrastructure for the End of Moore's Law", 2020. [arXiv:2002.11054](https://arxiv.org/abs/2002.11054). Full.
23. [C23] OMG, MDA Guide Version 1.0, 2003. [PDF](#). Full.
24. [C24] M. Petre, "UML in practice", ICSE 2013. [author copy](#). Full.
25. [C25] Protocol Buffers overview. protobuf.dev. Full.
26. [C26] Lumos introduction. docs.lumos-lang.org. Full.
27. [C27] Forge. github.com/Sanjith-Shan/Forge. Full (README).
28. [C28] Faber Romanus. github.com/faberlang/faber. Full (README).
29. [C29] Intent. github.com/lhaig/intent. Full (README).
30. [C30] NanoLang. github.com/jordanhubbard/nanolang. Full (README).

31. [C31] J. W. Backus et al., "The FORTRAN Automatic Coding System", 1957. [Computer History Museum copy](#). Full.
32. [C32] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis", ACM TOPLAS 2(1), 1980. [doi:10.1145/357084.357090](#). Full.
33. [C33] B. Böckeler, "Understanding Spec-Driven-Development: Kiro, spec-kit, and Tessl", [martinfowler.com](#), 15 October 2025. [martinfowler.com](#). Full.
34. [C34] S. Bursuc, T. Ehrenborg, S. Lin et al., "A benchmark for vericoding: formally verified program synthesis", 2025. [arXiv:2509.22908](#). Full.
35. [C35] X. Leroy, "Formal verification of a realistic compiler", CACM, 2009. [PDF](#). Full.
36. [C36] G. Klein et al., "seL4: Formal Verification of an OS Kernel", SOSP 2009. [PDF](#). Full.
37. [C37] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat and S. Seshia, "Combinatorial Sketching for Finite Programs", ASPLOS 2006. [PDF](#). Full.
38. [C38] T. Hubert et al., "Olympiad-level formal mathematical reasoning with reinforcement learning", Nature, 2025. [Nature](#). Full.
39. [C39] C. Sun et al., "Clover: Closed-Loop Verifiable Code Generation". [arXiv:2310.17807](#). Full.
40. [C40] C. Yang et al., "AutoVerus". [arXiv:2409.13082](#). Full.
41. [C41] Y. Sun, J. Liu, D. Kroening and J. Xue, "Agentic Model Checking", 2026. [arXiv:2605.21434](#). Full.
42. [C42] A. Cunha, "Verified LLM-Driven Synthesis for Concept Design", 2026. [arXiv:2607.15718](#). Full.
43. [C43] Vera. [github.com/aallan/vera](#). Full (README).
44. [C44] Vow. [github.com/vow-lang/vow](#). Full (README).
45. [C45] Thermite. [github.com/dollspace-gay/Thermite](#). Full (README).
46. [C46] Decision models as a class: see the decision-head section above and its sources [B7; B8; B12].
47. [C47] J. Schmidhuber, "Goedel Machines: Self-Referential Universal Problem Solvers Making Provably Optimal Self-Improvements", TR IDSIA-19-03 (v5, 2006). [arXiv:cs/0309048](#). Full.
48. [C48] E. Yudkowsky and M. Herreshoff, "Tiling Agents for Self-Modifying AI, and the Löbian Obstacle", MIRI early draft, 7 October 2013. [PDF](#). Full.
49. [C49] G. Wang et al., "Voyager: An Open-Ended Embodied Agent with Large Language Models", 2023. [arXiv:2305.16291](#). Full.
50. [C50] E. Zelikman, E. Lorch, L. Mackey and A. T. Kalai, "Self-Taught Optimizer (STOP): Recursively Self-Improving Code Generation", COLM 2024. [arXiv:2310.02304](#). Full.
51. [C51] J. Zhang, S. Hu, C. Lu, R. Lange and J. Clune, "Darwin Gödel Machine: Open-Ended Evolution of Self-Improving Agents", ICLR 2026. [arXiv:2505.22954](#). Full.
52. [C52] A. Novikov et al., "AlphaEvolve: A coding agent for scientific and algorithmic discovery", Google DeepMind white paper, 2025. [arXiv:2506.13131](#). Full.
53. [C53] J. Bader, A. Scott, M. Pradel and S. Chandra, "Getafix: Learning to Fix Bugs Automatically", OOPSLA 2019. [arXiv:1902.06111](#). Full.
54. [C54] K. Etemadi et al., "Sorald: Automatic Patch Suggestions for SonarQube Static Analysis Violations", 2021 (v2 2022). [arXiv:2103.12033](#). Full.

55. [C55] K. Liu et al., "TBar: Revisiting Template-based Automated Program Repair", 2019. [arXiv:1903.08409](#). Full.
56. [C56] R. Landauer, "Irreversibility and heat generation in the computing process", 1961. [doi:10.1147/rd.53.0183](#). Full.
57. [C57] C. H. Bennett, "Logical Reversibility of Computation", 1973. [doi:10.1147/rd.176.0525](#). Full.
58. [C58] L. Chen, M. Zaharia and J. Zou, "FrugalGPT", 2023. [arXiv:2305.05176](#). Full.
59. [C59] M. H. Erol, B. El, M. Suzgun, M. Yüsekönül and J. Zou, "Cost-of-Pass: An Economic Framework for Evaluating Language Models", ICLR 2026. [arXiv:2504.13359](#). Full.
60. [C60] S. Kapoor, B. Stroebl, Z. S. Siegel, N. Nadgir and A. Narayanan, "AI Agents That Matter", 2024. [arXiv:2407.01502](#). Full.
61. [C61] H. Pan, L. Huang, X. Zhou, M. Xing and Y. Wu, "Toward Compiler World Models: Learning Latent Dynamics for Efficient Tensor Program Search", 2026. [arXiv:2606.09312](#). Full.

Not carried over from the survey, because the primary source was not obtained or did not support the claim: Smalltalk image claims, Darklang hashing, Pantograph and Grove, Codong, Lume, Magpie, Codex (Tedrow), Spec IR, Boruna, d-raft, Sockudo, DBOS, P language compilation to C, UNCOL and ANDF, Thrift and OpenAPI, Pact, Aver, Prove, reqlan, translation validation (Pnueli), SyGuS, Futamura and Turchin, PAR, DéjàVu and Gabel and Su, Simon and Kadane, and the comparative scoreboard.

1. History: seventy years of "say what you want, get the program"

The wish behind this topic is old. What has changed is which parts of it are now cheap. The timeline below lists the main attempts; the text after it says what each one achieved and where it stopped.

 Timeline from FORTRAN in 1954 through deductive synthesis, program derivation, fourth-generation languages and CASE tools, the Fifth Generation project, No Silver Bullet, refinement methods and Model Driven Architecture, verified systems CompCert and seL4, large language model code generation, constrained decoding and verifier loops, to the open question of learned proposers with deterministic checkers

Figure 9. Attempts to turn a statement of intent into a working program. CASE means computer-aided software engineering; UML means Unified Modeling Language; Z, B and VDM (Vienna Development Method) are mathematical specification methods; Coq is a proof assistant. Green: machine-checked systems. Amber: learned models. Blue: the question this topic studies. [Open full size · PNG](#).

- **Automatic programming, 1950s.** FORTRAN (FORMula TRANslation) was presented as "The FORTRAN Automatic Coding System". Its authors wrote that its purpose was to let the IBM 704 "code problems for itself and produce as good programs as human coders (but without the errors)" (Backus et al., 1957, p. 188; PDF p. 2). Example: a scientist wrote a formula; the system wrote the machine code. Today that is called a compiler, not automatic programming.
- **Deductive synthesis, 1960s-80s.** Simon's heuristic compiler (1963) treated programming as heuristic problem solving: a means-end search, built on the General Problem Solver (GPS), that writes routines from descriptions of the state before and after, by "selective trial-and-error search" (Simon, RAND RM-3588-PR, 1963, Summary, p. vii). Manna and Waldinger (1980) treated a program as something extracted from a constructive proof that a specification can be met: "From such a proof, a program to compute z can be extracted" (Manna and Waldinger, STAN-CS-78-690, 1978, Specification section; the same sentence appears in the TOPLAS article, 1980, p. 92); both versions note that "no implementation exists" (report, Motivation, p. 2; TOPLAS, p. 91). The JACM article states Simon's conclusion directly: "compiling tasks--at least simple compiling tasks--can fruitfully be viewed as problem-solving tasks" (Simon, 1963, Summary, p. 505), and it makes no practical claim: "no claims will be made for the heuristic programming procedures described here as practical approaches to the construction of compilers" (Introduction, p. 493). Manna and Waldinger observed that theorem proving had been "abandoned as an approach to program synthesis when the development of sufficiently powerful automatic theorem provers appeared to flounder" (TOPLAS, 1980, p. 120). All four texts were read in full.
- **Program derivation, 1975-86.** Dijkstra introduced guarded commands "as a building block" for programs and presented "the formal derivation of programs" from conditions on the final state (Dijkstra, EWD 472, abstract). Bird's calculus of lists (the Bird-Meertens formalism) is "used to derive efficient solutions" by rewriting with algebraic identities (Bird, 1986, PDF p. 3). Example: for Gries's problem "to compute the minimum of the sums of all segments" of a list, the direct program takes $O(n^3)$ steps, and rewriting with the calculus yields "a linear time algorithm" (Sec. 4.2, PDF p. 34).
- **Fourth-generation languages, CASE tools and expert systems, 1980s.** Report generators, form builders and diagram tools promised programs from screens and pictures. Brooks reviewed graphical programming, automatic programming and expert systems in 1986 and argued that none of them "promises even one order-of-magnitude improvement" (Brooks, 1986, PDF pp. 1, 8-11). Kemerer reported, citing an earlier study, that "one year after introduction, 70 percent of CASE tools and techniques are never used" and that "the first project written with an integrated CASE tool typically fails to deliver improved results"; he attributes part of the abandonment to a learning curve that dips below the old technology before it rises (Kemerer, 1992, pp. 23-24, Fig. 1).
- **Japan's Fifth Generation Computer Systems project, 1982-92.** A national effort built on logic programming in the Prolog family; ICOT spent "about 54 billion Yen, that is some 450 million US\$" over ten years (van de Riet, 1993, Sec. 1, p. 83). One goal in the 1981 plan was "to lessen the burden of software generation in order that a high level

requirements specification is sufficient for automatic processing" (Sec. 3, p. 85). The parallel inference machine met its 100 MLIPS target, but van de Riet scores "break-through in architecture" and "break-through in software" as failures and judges technical impact "small" (Sec. 6, p. 103); on the effort needed to program the system, "there is almost no evidence" (Sec. 4.1, p. 86).

- **Refinement and model-driven methods, 1980s-2000s.** Z, B and VDM refine a mathematical specification step by step toward code. The Model Driven Architecture guide of the Object Management Group "specifies three viewpoints on a system, a computation independent viewpoint, a platform independent viewpoint and a platform specific viewpoint" (OMG, 2003, Sec. 2.2.5, PDF p. 13). Example: draw a class diagram once and generate Java and SQL (Structured Query Language) from it. Hutchinson et al. note that MDE "has been developed largely without the support of empirical data" (2011, Sec. 1, p. 471); among 250+ practitioners surveyed, 74% said MDE "requires them to carry out significant extra training" (Sec. 4.3.1, p. 475), and one interviewee reported 4 to 8 times productivity internally while telling outsiders "1.5 to 2" (Sec. 5.2.6, p. 478). The authors call their results "interesting and suggestive rather than conclusive" (Sec. 3.1, p. 473).
- **Machine-checked systems, 2008-09.** CompCert is a C compiler whose "proof of semantic preservation" was carried out in the Coq proof assistant (Leroy, 2009, abstract, PDF p. 1). seL4 is a microkernel of "8,700 lines of C code" with a machine-checked "proof of functional correctness" (abstract, PDF p. 1); the proof cost "about 20 py" (person-years) (Klein et al., 2009, Sec. 5.2, PDF p. 13). Coq can extract a proved function into ordinary code in "functional languages such as Ocaml, Haskell or Scheme" (Letouzey, 2008, Sec. 1, PDF p. 1); the extraction step itself was not yet mechanically verified, and its correctness rested on paper proofs (Sec. 6, PDF p. 10).

2. Why the earlier attempts stalled

- **Essential versus accidental complexity.** Brooks separated "essential tasks, the fashioning of the complex conceptual structures that compose the abstract software entity", from "accidental tasks, the representation of these abstract entities in programming languages" (abstract, PDF p. 1). He quoted Parnas: "automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer" (PDF p. 9), and wrote that "even perfect program verification can only establish that a program meets its specification" (PDF p. 11) (Brooks, 1986). Example: a better notation for a payroll rule does not decide what the rule should be.
- **Natural language is ambiguous.** Dijkstra argued that formal texts are, "when you come to think of it, an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid" (Dijkstra, EWD 667, para. 6). Example: "sort the users by name and date" has at least two readings.

- **Specification costs as much as the program.** In seL4 the abstract specification took "about 4 person months" and the proof about 20 person-years, against "a total cost of 2.2 py including the Haskell effort" to build the kernel itself (Klein et al., 2009, Sec. 5.2, PDF pp. 12-13). The 20 py includes about 9 py of research and reusable proof tooling; the seL4-specific proof "was 11 py" (PDF p. 13).
- **Combinatorial search.** Searching all programs grows exponentially with program length. DreamCoder addresses this by learning a library of reusable pieces and a neural network "to guide the search for programs" in a "wake-sleep" loop (Ellis et al., 2021, abstract, PDF p. 1); the paper calls such search "intractable for general-purpose algorithms, because of the combinatorial nature of the search space" (introduction, PDF p. 2); the supplement repeats that finding programs "is difficult because of the infinitely large, combinatorial search landscape" (supplement S4.1, PDF p. 42 of the version with supplement), the same kind of search problem that limited deductive synthesis.
- **Brittleness and adoption.** Diagram-to-code tools generated code that was then hand-edited, and the model and the code drifted apart. Petre reports from earlier surveys that "the biggest perceived problem of model-centric approaches is keeping the model up-to-date with the code (68% agreement)" (Forward and Lethbridge, as reported by Petre, 2013, author copy p. 3); among Petre's own 50 informants, 35 did not use UML (Unified Modeling Language) and 3 used automated code generation (pp. 4-5). Adoption studies of CASE and of model-driven engineering are listed in the ledger as not read, so no figures from them are quoted.
- **Opinion.** The common thread is that each approach fixed one cost (notation, proof, generation) while another stayed manual: writing the specification, searching for the program, or checking the result.

3. Current attempts

A large language model (LLM) is a neural network trained to predict the next token (a word piece) of text, including program text.

- **Error rates.** HumanEval is a set of 164 hand-written Python problems checked by unit tests; pass@k is the share of problems solved by at least one of k samples. Codex solves 28.8% with one sample and 70.2% with 100 samples (abstract; Table 1 gives 28.81% pass@1 and 72.31% pass@100 for Codex-12B, PDF p. 7), and Codex-S, generating 100 samples per problem, reaches 77.5% "by selecting the sample that passes the unit tests", against 44.5% when picking the sample with the highest mean log-probability (Chen et al., 2021, Fig. 1 and Sec. 1, PDF p. 2). That selection uses the benchmark's own unit tests as "an oracle with prior knowledge of the unit tests" (Sec. 3.3, PDF p. 5). Pass rate drops "by roughly a factor of 2-3" for each additional chained component in a synthetic task description (Sec. 6, Fig. 11, PDF p. 10). SWE-bench is 2,294 real GitHub issues from 12 Python repositories; with BM25 (a keyword search ranking) retrieval, Claude 2 resolves 1.97% (1.96% in the text of Sec. 5) and Claude 3 Opus 3.79%, and only 43.07% and 46.56% of their patches even apply (Jimenez et al.,

2024, Table 5), each from a single greedy-decoded patch per issue. AlphaCode generated a very large number of samples per problem and filtered them "based on program behavior", reaching "a ranking of top 54.3%" in contests with more than 5,000 participants (Li et al., 2022, abstract, PDF p. 1). The authors put this at roughly the level of the median competitor, with at most 10 submissions per problem picked from up to millions of samples (Secs. 5.1 and 9).

- **Nondeterminism.** Asking ChatGPT (gpt-3.5-turbo at its default temperature of 1) the same coding question five times, the share of problems with no test on which all five answers gave the same output was 75.76%, 51.00% and 47.56% on three datasets, and "setting the temperature to 0 does not guarantee determinism in code generation" (Ouyang et al., 2023, abstract, PDF p. 1). At temperature 0 those shares were still 43.64%, 27.40% and 18.29% (Sec. 4.2).
- **Constrained decoding.** Restricting each next token to those a grammar allows. Synchromesh's Constrained Semantic Decoding "guarantees that all sampled programs satisfies the implemented constraints", including "syntax, scope, typing rules" (Poesia et al., 2022, abstract and Sec. 1, PDF pp. 1-2). It does not prevent conceptual errors, where a valid program does the wrong thing (Sec. 4.2). Willard and Louf index a model's vocabulary against the states of a finite-state machine built from a regular expression, and sketch an extension to context-free grammars through pushdown automata, so the restriction "adds little overhead" (Willard and Louf, 2023, abstract and Secs. 3-4, PDF pp. 1-15). Their speed comparison is a single timed sample per length against one other library (Sec. 3.2).
- **Model plus verifier loops.** AlphaProof writes proofs in the Lean proof assistant, which checks each one, and with it the system "solved three out of the five non-geometry problems" at the 2024 International Mathematical Olympiad (Hubert et al., 2025, abstract, PDF p. 1). Experts formalized the problems in Lean by hand after release, and each of the three solutions took 2-3 days of test-time reinforcement learning, well beyond contest time limits (IMO 2024 section). LeanDojo pairs an LLM prover with premise retrieval in Lean; its ReProver proves 51.2% of held-out test theorems on a random split and 26.3% when the proofs need premises unseen in training (Yang et al., 2023, Table 2). For code, Clover checks consistency among Dafny code, documentation and formal annotations and "accepts 87% of the correct examples and rejects 100% of the adversarial incorrect" ones on its benchmark (Sun et al., 2024, Sec. 1, PDF p. 3). The 87% is 52 of 60 with up to 10 attempts per check (75% with one), and the incorrect examples are hand-made variants (Table 1). AutoVerus uses LLMs to write proofs for Rust code checked by the Verus verifier and "can automatically generate correct proof for more than 90%" of its 150 benchmark tasks (Yang et al., 2024, abstract, PDF p. 1). That is 137 of 150 with up to three runs, with the specifications supplied (Sec. 7.1, Table 2).
- **No-code and low-code tools.** These pair a fixed schema (forms, tables, workflows) with a runtime that interprets it. *Opinion:* they show that a closed program shape plus a deterministic runtime is practical for a narrow class of applications; their limit is how much the schema can express.

- **Energy-based models.** An energy-based model "capture[s] dependencies between variables by associating a scalar energy to each configuration"; inference means "finding configurations of the remaining variables that minimize the energy", with "no requirement for proper normalization" (LeCun et al., 2006, abstract, PDF p. 1). Low energy means "these fit together".
- **Joint Embedding Predictive Architectures (JEPA).** I-JEPA (the image version) predicts "the representations of various target blocks" from a context block, so that "unnecessary pixel-level details are potentially eliminated" (Assran et al., 2023, Sec. 1, PDF p. 2). Computing the loss on raw pixels instead of in latent space dropped one image benchmark score (linear probe on 1% of ImageNet) from 66.9 to 40.7, with the pixel-target model trained 800 epochs against 500 (Table 7, PDF p. 8); the authors present the representation-space explanation as a conjecture (Sec. 9). V-JEPA 2 (the video version) plans by minimizing an energy, the distance between a predicted future representation and a goal representation (Eq. 5, PDF p. 10), and reports "only 16 seconds per action" versus 4 minutes per action for a generative video model, while using 10× more samples per refinement step (Assran et al., 2025, Sec. 4.2, PDF p. 13; Table 3, PDF p. 15). The framing comes from LeCun's 2022 position paper, which states that a JEPA "does not actually predict y, but predicts the representation of y" (LeCun, 2022, Sec. 4.3, PDF p. 24).

4. Why now: the case for closed-form, deterministic, AI-proposed code

This section is opinion, built from the observations above. It is a case to be tested, not a result.

- **Checkers are cheap and trustworthy where the earlier attempts lacked them.** Using Csmith, a random program generator, the authors found that CompCert was "the only compiler we have tested for which Csmith cannot find wrong-code errors" after "about six CPU-years" of trying (Yang et al., 2011, Sec. 3.1, PDF p. 6). Machine-checked rules can carry the trust that a generator cannot. CompCert itself uses this pattern: its register allocator runs an unverified graph-coloring heuristic and checks each result with a verifier proved correct in Coq, because graph coloring is "easier to validate a posteriori than to directly prove correct" (Leroy, 2009, Sec. 4.2, PDF p. 7).
- **Deterministic semantics answer the nondeterminism problem.** If the generator is allowed to vary but the accepted artifact is checked and replayed exactly, run-to-run variation in the model stops mattering for the result.
- **Verifier-in-the-loop evidence exists.** AlphaCode's behavioral filtering, Codex-S's oracle test selection (77.5% versus 44.5% for log-probability ranking), Clover, AutoVerus and AlphaProof all show large gains when a check selects among proposals.
- **Learned predictors now work in latent space.** I-JEPA and V-JEPA 2 show that predicting a summary of the target, rather than every detail, can work well for those tasks: V-JEPA 2 had higher success on 4 of 5 robot tasks (tied on one) and planned about 15x faster

than a generative video model in its comparison (Table 3, PDF p. 15). In a program, details such as identifier spelling, formatting and target language can be filled in deterministically after the structure is chosen.

- **Token-by-token generation is expensive.** LLaMA 65B, sharded across 8 to 32 V100 GPUs with a 512-token output limit, used "about 3-4 Joules for a output token", counting GPU energy only (Samsi et al., 2023, Secs. III-C and IV-C, PDF pp. 4-6). In a study of 88 models, text-generation models averaged 0.047 kWh (kilowatt-hours) per 1,000 queries, each query generating only 10 new tokens (Table 2), and "multi-purpose, generative architectures are orders of magnitude more expensive than task-specific systems", partly because a generator chooses among "an entire vocabulary" (Luccioni et al., 2024, abstract, PDF p. 1; Sec. 4.1 and Table 2, PDF p. 6; Sec. 4.2.1, PDF p. 8).
- **Illustrative arithmetic only, mixing sources.** At 3-4 joules per token (Samsi et al.), a 300-token function costs about 0.9-1.2 kJ (kilojoules) per sample; at the 28.8% one-sample pass rate of Codex-12B (Chen et al.) that is roughly 3-4 kJ per test-passing function before counting test execution. These numbers come from different models and hardware. A proper test measures both on the same machine.

What could still fail

- **Expressiveness ceiling (H6).** If real programs need effects or nondeterminism the Form cannot state, the "all code" premise fails.
- **Intent may be the bottleneck.** A smaller output space does not fix underspecified intent. Signal: H1 and H2 hold while H3 fails.
- **Spec gaming.** The checker proves a Form is well-formed and meets stated properties, not that it matches what the user meant. Benchmark tasks carry behavioral traces checked by replay; the residual risk is labeled, not solved.
- **Corpus scarcity.** Signal: H3 flat as the corpus grows.
- **Collapse.** JEPAs need anti-collapse measures, such as the EMA (exponential moving average) target encoder of I-JEPA and V-JEPA 2; LeCun (2022) proposes VICReg-style information maximization instead (Sec. 4.5). A collapsed Form embedding makes H5 fail.
- **Measurement error.** Rootless counters (macmon, RAPL) cover package or SoC, not wall power; both arms use the same meter.
- **Baseline drift.** The quoted figures come from 2021-2025 systems; a fair test re-measures a current baseline on the same hardware.

5. One concrete instance: the Reality Code Engine

RCE is one closed program form among possible others; the questions above apply to any language with a small, checkable program shape. In RCE every program is a Form $P = (S, E, \delta, D, \lambda)$:

- **S, state:** the typed data the program holds, for example the cursor position and text buffer of an editor.
- **E, events:** the inputs it can receive, for example "key pressed" or "file opened".

- **δ (delta), transition function:** the rule giving the next state from the current state and an event.
- **D, derivations:** the recorded, hash-chained history of steps, which makes every run replayable bit for bit.
- **λ (lambda), output:** what the program shows or emits for a given state, for example the visible lines of text.

A checker accepts a Form or names the rule it breaks. Accepted Forms are proved against stated properties, replayed exactly, emitted as code in other languages and run on RCE's own engine, which also places work across hardware. The full description is on the [RCE topic page](#) and the public specification at github.com/openIE-dev/rce; nothing below depends on having read them.

RCE-JEPA is the working name for the task studied here: decoding stated intent into an accepted Form with a joint embedding predictor, leaving checking, proofs, derivation, emission and placement to RCE. The aim is the best performance per clock cycle and per joule. *Working hypothesis:* if all code can be encoded in the Form, all code can be produced by joint embedding or energy-based models over it. The hypotheses below are designed so that the evidence can refute this.

Defined quantities

- **Output space.** Free-form generation over vocabulary V with length L has raw space $|V|^L$. Ω_n (omega) is the set of valid Forms of size n (Form nodes); $\Omega_n^\vee \subseteq \Omega_n$ is the subset the checker accepts. Reported as $\log_2 |\Omega_n|$ bits, estimated by grammar counting and sampling.
- **Description length.** $b(P)$ is the bits to encode Form P canonically; $b(\text{code}) = \text{tokens} \times \log_2 |V|$ for the equivalent emitted source. Ratio ρ (rho) = $b(\text{code}) / b(P)$.
- **Verification cost.** $C_{\text{check}}(P)$: wall time, cycles and joules for the checker to decide P . Unit tests give C_{test} ; a passing test is evidence, not proof.
- **Acceptance probability.** $a = P(\text{checker accepts and task specification satisfied} \mid \text{intent})$, measured as pass@1 over the benchmark.
- **Joules per accepted program.** $J_{\text{acc}} = (E_{\text{model}} + E_{\text{check}} + E_{\text{retry}}) / N_{\text{accepted}}$, summed over a benchmark run, energy measured at the device with idle power subtracted.
- **Cycles per accepted program.** K_{acc} , the same in CPU (central processing unit) or GPU cycles where counters are readable.
- **AUROC** (area under the receiver operating characteristic curve): the probability that a randomly chosen accepted Form gets a lower energy than a randomly chosen rejected one; 0.5 is chance, 1.0 is perfect.

Argument (opinion, to be tested). A Form has five typed slots with fixed composition rules. Constraining output to Ω_n removes most of $|V|^L$ before the model is consulted, so fewer bits are predicted per program ($\rho > 1$), the checker gives a definite accept or reject in place of a

probabilistic test, and rejections are cheap and structured, so retries are guided rather than blind.

Falsifiable hypotheses H1 to H13

All thirteen hypotheses in numerical order. H1 to H7 test the RCE-JEPA thesis directly, H8 to H10 test the "Laws of Coding" thesis above, and H11 to H13 test the proposer (the H3 control is in the proposer section above; the falsifiers RJ-F1 to RJ-F5 follow this table). All measured on the benchmark below, same hardware, idle-subtracted joules. Baseline: an next-token code LLM of matched parameter count generating source in a target language, judged by that language's tests plus the RCE spec where translatable.

ID	Hypothesis	Metric	Supported if	Disproved if
H1	Form space is much smaller than source space for the same programs	$\rho = b(\text{code})/b(P)$, median over corpus	$\rho \geq 5$	$\rho < 2$
H2	Checker is cheap relative to generation	$C_{\text{check}} / E_{\text{model}}$ per candidate (joules)	≤ 0.05	≥ 0.5
H3	Representation-space prediction reaches useful acceptance	pass@1 accepted Forms on held-out intent-to-Form tasks	$\geq$ baseline pass@1 on the same tasks	$< 0.5 \times$ baseline
H4	Energy per accepted program falls	$J_{\text{acc}}(\text{RCE-JEPA}) / J_{\text{acc}}(\text{baseline})$	≤ 0.1	≥ 1
H5	Energy ranking concentrates mass on valid Forms	top-1 type-check rate; AUROC of E for accept vs reject	≥ 0.8 top-1 valid; AUROC ≥ 0.85	AUROC ≤ 0.6
H6	The reference apps are encodable	share of the 6 reference apps plus RCEdb fully expressed as Forms that pass the checker and replay bit-identically	7 of 7	any app requires an escape hatch outside the Form
H7	Security is model-independent	adversarial Forms (fuzzed and model-generated) accepted by the checker that violate a stated invariant	0	≥ 1 confirmed
H8	Determinism: every accepted Form replays to a bit-identical trajectory and output	mismatches over at least 10,000 recorded event streams on at least 3 different machines and operating systems	0	a single mismatch
H9	Cost is predicted before execution	median error of the cost-table prediction against measured energy (J, named meter) for a	$\leq 20\%$	$> 50\%$

ID	Hypothesis	Metric	Supported if	Disproved if
		stated machine, over the test families		
H10	Learning on the quotient saves data	training examples a predictor of canonical Forms needs to reach the same test acceptance rate as one of equal size trained on surface code	at most half (a saving of 2 times or more)	a saving below 1.25 times
H11	Energy is monotone in edit distance	Share of test families in which energy E rises with edit distance 1 to k from an accepted fit	$\geq 80\%$ of families	$< 80\%$ of families
H12	The decision head is calibrated on gate outcomes	ECE per decision type, after 10^4 receipts	$ECE \leq 0.1$ and below a frequency table over shape hashes	$ECE > 0.2$, or no better than the frequency table
H13	A discrete latent plans better than a continuous one over long edit sequences	Planning success at edit-sequence lengths 1, 5 and 20, discrete arm against continuous SIGReg arm	Discrete arm higher at length 20	Neither arm reaches the H3 threshold

Falsifiers RJ-F1 to RJ-F5 for the premise itself

Each falsifier is an observation that would defeat part of the premise; several are stated in terms of the hypotheses above (RJ-F3 on H3, RJ-F4 on H11).

ID	Observation	What it defeats
RJ-F1	Lift coverage plateaus below 80% on any of the five target languages after the construct list is exhausted	The claim that all code can be emitted from RCE, and with it the scope of autonomous coding over Forms
RJ-F2	Writing $\sigma(l)$ for the 7 reference applications costs more person-hours than writing their Forms by hand	The claim that closure removes, rather than moves, the specification cost
RJ-F3	H3 fails while the token arm passes at the same gate	Latent proposal over Form space; the gate still stands, the JEPA half does not
RJ-F4	H11 fails and search does not reach accepted Forms within the beam budget	Energy-guided planning in Form space
RJ-F5	The decision head's calibration never beats a frequency table over shape hashes	The decision-model leg; it adds nothing over promoted-handler statistics

The thesis is disproved overall if H4 fails (no joule advantage per accepted program) or H6 fails (the Form cannot express the reference apps). Either alone is sufficient.

What will be delivered

Scope, organised as parallel workstreams that meet at the evaluation harness. Every deliverable either produces Forms, produces models that propose Forms, or measures the result.

- **W1. Form corpus and dataset card.** Forms paired with intent statements and checker verdicts, from hand-written Forms, reference-app Forms and checker-verified mutations; the card covers provenance, licence, size distribution, negative generation, known gaps and contamination controls; canonical serialization for H1.
- **W2. Encoder and predictor.** Intent encoder, EMA Form target encoder, predictor in representation space and energy head $E(\text{intent}, P)$, with anti-collapse measures, an embedding-space versus token-space loss ablation, and model cards with measured training energy.
- **W3. Checker-constrained decoder.** Grammar-constrained candidate Forms ranked by energy and submitted to the checker, rejection diagnostics fed back under a fixed retry budget; the checker is the sole acceptance authority with no bypass path to the engine.
- **W4. Intent-to-Form benchmark.** Unit and composite tasks, the reference apps (game engine, text editor, video player, chat app, browser, terminal) and RCEdb; each task has intent, reference Form, replay traces and an equivalent source-language task for the baseline; held-out split.
- **W5. Evaluation harness with energy in joules.** Both arms on identical hardware and tasks; $\text{pass}@1$, $\text{pass}@k$, ρ , C_{check} , J_{acc} , K_{acc} , retries per acceptance, energy-ranking AUROC; rootless counters, idle subtracted, raw logs kept, unavailable counters reported rather than estimated.
- **W6. Research article.** This case expanded with results for H1 to H13, including negative results, with figures, tables and reproducibility instructions.
- **W7. Open artifacts.** Public: the Form specification slice, checker acceptance rules, benchmark tasks and traces, harness, dataset and model cards, and weights where licences allow. Private: engine source, placement logic and emitter internals. Anything needed to reproduce a reported number is public, or the number is reported as non-reproducible.
- **W8. Demo.** Stated intent to predicted Form to checker verdict (including a shown rejection) to engine replay and emitted code in two languages, with an on-screen joule reading.

Status

Research topic open. The gate, replay, projections and energy accounting are described in the RCE paper; no number from that paper is used here as evidence. The phased pilot plan runs from the data engine (Phase 0) to convergence (Phase 5), each phase preregistered with exit criteria. Phase 0 has started: version 1 of the Form corpus (W1) and the first measurement of H1 are published (23 October 2026; see the data section of the proposer above), with H1 supported on the hand-written population at a median ρ of 9.31. H8 has a first recorded run,

preregistered before it: 10,500 seeded event streams over the example programs, traced and hashed on six targets (x86_64, aarch64, s390x big-endian and wasm32; Linux, macOS and Windows) by the repository's replay-hash parity workflow, with 0 mismatching lines against the reference; the run, the targets and the limits are in the experiments repository under `rce/bench/determinism/`. H7 has a first run on its fuzzed arm, preregistered before it: 1,275 checker-accepted mutants of the Form corpus over two runs were checked against five stated invariants (the hash chain, atomicity of refusal, determinism of two runs, re-acceptance by the gate, termination under fuel) with 0 violations; the model-generated arm does not exist yet (`rce/bench/adversarial/`). The proposer, the decision head and the hypotheses H2 to H6 and H9 to H13 remain planned.

References

Each source carries a reading label. **Full:** read in full. **Partial:** the full text was retrieved and searched for the quoted passages, which were checked in their surrounding text, but the whole work was not read. **Unavailable:** not obtained. "PDF p." gives the page in the linked copy; Sec. gives the section.

- Assran et al. 2023. Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture. CVPR. [arXiv:2301.08243](#). Full.
- Assran et al. 2025. V-JEPA 2. [arXiv:2506.09985](#). Full.
- Backus et al. 1957. The FORTRAN Automatic Coding System. Western Joint Computer Conference. [Computer History Museum copy](#). Full.
- Bird 1986. An Introduction to the Theory of Lists. Oxford PRG-56. [PDF](#). Full.
- Brooks 1986. No Silver Bullet: Essence and Accident in Software Engineering. [PDF](#) (page numbers refer to this copy). Full.
- Chen et al. 2021. Evaluating Large Language Models Trained on Code. [arXiv:2107.03374](#). Full.
- Dijkstra. EWD 472, Guarded commands, non-determinacy and formal derivation of programs. [Transcription](#). Full.
- Dijkstra. EWD 667, On the foolishness of "natural language programming". [Transcription](#). Full.
- Ellis et al. 2021. DreamCoder. PLDI. [arXiv:2006.08381](#); [version with supplement](#). Full (main text and supplement S1 to S4.8, PDF pp. 23-68, read in full).
- Hubert et al. 2025. Olympiad-level formal mathematical reasoning with reinforcement learning (AlphaProof). Nature. [Nature](#). Full.
- Jimenez et al. 2024. SWE-bench. ICLR. [arXiv:2310.06770](#). Full.
- Klein et al. 2009. seL4: Formal Verification of an OS Kernel. SOSP. [PDF](#). Full.
- LeCun et al. 2006. A Tutorial on Energy-Based Learning. [PDF](#). Full.
- Leroy 2009. Formal verification of a realistic compiler. CACM. [PDF](#). Full.
- Letouzey 2008. Extraction in Coq: an Overview. CiE. [PDF](#). Full.
- Li et al. 2022. Competition-Level Code Generation with AlphaCode. [arXiv:2203.07814](#). Full.

- Luccioni, Jernite, Strubell 2024. Power Hungry Processing. FAccT. [arXiv:2311.16863](#). Full.
- OMG 2003. MDA Guide Version 1.0. [PDF](#). Full.
- Ouyang et al. 2023. An Empirical Study of the Non-determinism of ChatGPT in Code Generation (v1 title: LLM is Like a Box of Chocolates). [arXiv:2308.02828](#). Full.
- Poesia et al. 2022. Synchromesh. ICLR. [arXiv:2201.11227](#). Full.
- Samsi et al. 2023. From Words to Watts. IEEE HPEC. [arXiv:2310.03003](#). Full.
- Sun et al. 2024. Clover: Closed-Loop Verifiable Code Generation. [arXiv:2310.17807](#). Full.
- Willard and Louf 2023. Efficient Guided Generation for Large Language Models. [arXiv:2307.09702](#). Full.
- Yang et al. 2011. Finding and Understanding Bugs in C Compilers (Csmith). PLDI. [PDF](#). Full.
- Yang et al. 2023. LeanDojo. NeurIPS. [arXiv:2306.15626](#). Full.
- Yang et al. 2024. AutoVerus. [arXiv:2409.13082](#). Full.
- LeCun 2022. A Path Towards Autonomous Machine Intelligence, version 0.9.2. [OpenReview](#). Full.
- Simon 1963. Experiments with a Heuristic Compiler. Journal of the ACM 10(4), 493-506. [DOI 10.1145/321186.321192](#). Full.
- Manna and Waldinger 1980. A Deductive Approach to Program Synthesis. ACM TOPLAS 2(1), 90-121. [DOI 10.1145/357084.357090](#). Full.
- van de Riet 1993. An overview and appraisal of the Fifth Generation Computer System project. Future Generation Computer Systems 9, 83-103. [DOI 10.1016/0167-739X\(93\)90002-7](#). Full.
- Kemerer 1992. How the Learning Curve Affects CASE Tool Adoption. IEEE Software 9(3), 23-28. [DOI 10.1109/52.136161](#). Full.
- Hutchinson, Whittle, Rouncefield and Kristoffersen 2011. Empirical Assessment of MDE in Industry. ICSE, 471-480. [DOI 10.1145/1985793.1985858](#). Full.
- Petre 2013. UML in practice. ICSE. [DOI 10.1109/ICSE.2013.6606618](#); [author copy](#), [Open Research Online](#) (page numbers refer to this copy). Full.
- Simon 1963. Experiments with a Heuristic Compiler. RAND RM-3588-PR. [Internet Archive](#); a memorandum related to but distinct from the JACM article. Full (prose pp. iii-63; program listings in Appendices A and B scanned).
- Manna and Waldinger 1978. A Deductive Approach to Program Synthesis. Stanford STAN-CS-78-690. [Report version](#), which predates the TOPLAS article. Full.