Competence Without Intelligence: Synthetic Data, Verifiable Rewards, and Where AI Improves

Luís Seabra

Senior Machine Learning Engineer, Protegrity, 2026/04/18

Abstract

In 2026, AI is transforming software engineering substantially faster than predicted. The mechanism behind it is more revealing than any benchmark score or lab announcement, and it has little to do with model capability. It is about data, and specifically about whether that data can be verified.

This paper argues that the dramatic progress visible in AI coding agents is best understood as a data problem. The challenge is not generating synthetic training examples at scale but knowing which of those examples are actually correct. When outputs can be graded as correct or incorrect without human judgment, synthetic data becomes trustworthy and reinforcement learning becomes productive. This is the verification function, and its availability determines where synthetic data pipelines can succeed and where they cannot. Software engineering uniquely satisfies this condition through executable test suites that return a clean binary signal. Most other domains do not, and the more useful question for 2026 and onwards is where else reliable verification functions may be engineered, and what synthetic data pipelines could follow.

I. Every Past Has Its Future — And Its Data

There is a pattern to how we imagine artificial intelligence, where we always project what we expect it to be in a different direction than what eventually happens. Every generation of AI research has produced a vision of the future, and, as so often happens with science fiction, it does not age well in its specifics. The influence tends not to flow from the prediction but from wherever the previous wave happened to find solid ground — and solid data.

In the 1980s, the dominant paradigm was symbolic AI and expert systems, where hand-crafted rules would encode the knowledge of specialists, promising machines that could reason from knowledge, making doctors or lawyers obsolete. The data of that era was rules: these were curated, brittle, and expensive to maintain. Soon these systems hit walls that no amount of rule-writing could overcome. In the early 2010s, the explosion of deep convolutional neural networks, training on ImageNet and running on GPUs, redirected the field toward perception layers with an initial Computer Vision focus. The data of that era was labelled images, assembled at industrial scale through crowdsourcing and annotation pipelines. AI was now mostly imagined around how machines would see and recognize the world, toward understanding it. Radiologists would be the first profession to go. Self-driving vehicles were the flagship example with Level 5 autonomy, repeatedly, a few years away.

Between 2017 and 2020, the transformer architecture with its attention mechanism reshuffled projections again. BERT, GPT and their successors demonstrated that language could be modelled at scale, allowing useful, general-purpose next-token predictors as building blocks of natural language processing pipelines. The data of this era was the Internet itself: billions of documents, scraped, filtered, and fed into training runs of unprecedented scale. Simply by being prompted in the right way, models could do zero-shot and few-shot adaptation to tasks they had not been explicitly trained on. The predictions adjusted accordingly: universal knowledge assistants, AI tutors, agents managing your calendar and finances. Then in late 2022 ChatGPT brought these capabilities to a mass audience in a single product, and the excitement went off the charts. It was genuinely useful for drafting, researching, explaining. This cross-domain usefulness and its fluency signaled to many that something qualitatively new had arrived. Artificial General Intelligence, or AGI, was weeks away, or months, or perhaps already here depending on whom you asked. The hype around AI agents as personal companions, systems that knew you and would act for you autonomously in the world, reached a rhetorical peak through 2024, with autonomous agents placed at the top of Gartner's Hype Cycle for Artificial Intelligence [Gartner2024].

It is now 2026. While prominent voices in the field claim AGI is imminent or already functionally here, other equally prominent voices remain openly skeptical, arguing the benchmarks are saturating faster than the capabilities are generalizing. In practice, and despite the positive spin, personal AI companions have not yet arrived, save for a handful of DIY enthusiasts and entrepreneurs wiring something up themselves. AI doctors, including radiologists, and AI lawyers remain tools to be used by human practitioners, not replacements for them. Self-driving in a scalable way is proving harder and more expensive than hoped for. And yet the field that AI is actually transforming is the one closest to home for its creators: software engineering.

This paper asks why, and the answer has less to do with model capability than with synthetic data, and specifically with whether it can be verified at scale.

II. On Intelligence

Before claiming to know whether AI is generally intelligent, it is worth framing what intelligence actually is, a question that has occupied philosophers and cognitive scientists for decades, and which I do not intend to address here.

Textbook definitions involve some combination of learning from experience, applying knowledge to novel situations, reasoning under uncertainty, and, crucially, the capacity to pursue goals in a dynamic, complex environment. A useful distinction, drawn by the psychologist Kahneman, is between fast, automatic cognitive processes, pattern recognition, habitual responses, instinct, and slower, deliberate reasoning [Kahneman2011]. A large proportion of what we call thinking, in everyday human life, does not actually involve what we would want to call intelligence. Looking up a word, recalling a social convention, responding to a question with a rehearsed answer: these draw on prior learning and serve adaptive functions, but they are not what defines intelligence as a distinctively human capability. They are closer to sophisticated, learned instinct than to generative reasoning.

This distinction matters when evaluating current AI systems. Auto-regressive models learn a probability distribution over training text and produce the statistically likely continuation of whatever has been said so far. This description understates what they can visibly accomplish: extract concepts, follow novel instructions, and generalize across situations they were never explicitly trained on. The outputs are too coherent, too contextually sensitive, too useful for a simple dismissal. And yet they are, in essence, a very fast and very well-calibrated instinct engine running on the accumulated residue of human reasoning, compressed and made retrievable.

Competence is not the same as intelligence, a distinction which I do not think is merely cosmetic. What LLM-powered AI has learned is better described as the shape of intelligent output, not the process that produces it. Their first instinct is to answer, comply, and execute. They do not question the premise, interrogate the context, or push back and try to change the world around them. They are machines that answer questions, not sentient beings that ask them.

III. The Data Problem and why Synthetic works for Software

The reason these engines are succeeding in software is not that they are intelligent. It is that software gives them something most domains cannot: a way to know, automatically and without human judgment, when they got it right. To answer why, start with data.

Something remarkable is happening in software engineering in 2026. In February 2025, Karpathy coined the term 'vibe coding' [Karpathy2025] to describe fully delegating code generation to an AI, completely sidestepping any human reviewing. It was originally conceived for throwaway weekend projects and widely dismissed as a toy, something fit for hobby apps and demos but with no place in production systems where real consequences follow from bad code. Then something unexpected happened: it worked. Not perfectly, not safely by traditional standards, but well enough that developers started using it as a framework for real work, then teams, then companies. Within a year the conversation had moved from whether vibe coding was serious to how to scale it. Coding agents are now decomposing requirements, making architectural decisions, writing tests, catching their own errors, and shipping working software, all with minimal human revision or code input. The term 'agentic engineering' is on the rise [Karpathy2026] to support this movement.

What is emerging is genuine competence at the task, something which may be uncomfortable for those who have always thought of software engineering as a craft. The question I find important is not whether this is a signal of approaching general intelligence, but what it is about software engineering that made it the first domain to be transformed.

The Data Wall

The scaling era of internet-scraped text is approaching its limits. The stock of high-quality human-written text on the internet is finite, and frontier models have consumed most of what is worth consuming. Projections suggest that at current training scales,

models could exhaust the available supply of unique, useful web text within this decade [VillalobosEtAl2024]. The returns from scaling pre-training data are diminishing, and the field has been looking for alternatives.

Synthetic data, text generated by models or automated pipelines rather than by humans, is the most promising alternative, but it comes with a catch. Training models on their own generated data, if done naively, leads to what is called model collapse: a gradual narrowing of the output distribution, where the model becomes increasingly confident about an increasingly restricted range of responses [ShumailovEtAl2024]. The outputs grow stale, rare knowledge fades, and the model converges on the average. You end up with more data and a worse model.

Generating useful synthetic data requires more than prompting a model and collecting outputs. In practice, effective pipelines seed generation with real examples to anchor diversity, use structured templates for prompting strategies, use diverse base models, including fine-tuned ones, and apply filters to reach the edges of the distribution and ensure rare but important cases are not crowded out by the modal majority. Without these safeguards, collapse risk is highest: a model asked to generate its own training data will tend toward confident, common outputs and systematically underrepresent the long tail. Verification addresses this not by constraining what gets generated, but by selecting or labeling what gets used for training. Rare correct examples survive the verification filter just as common ones do, which preserves distributional coverage in a way that generation alone cannot guarantee.

The response from the research community has been telling. HuggingFace's FineWeb-Edu, for instance, filters a 15 trillion token corpus down to 1.3 trillion tokens using classifiers trained on synthetic annotations [LozhkovEtAl2024]. Synthetic data pipelines are now used at every stage, to filter and generate pretraining data using model-based quality signals [NiklausEtAl2026]; and to generate RL training data graded by hard verification functions. The latter is what is driving the most consequential AI improvements of recent years.

Synthetic data generation is cheap but verification is hard. Any capable LLM can produce training examples at scale, for almost any domain, at negligible cost. Knowing which of those examples are actually correct, and worth learning from, without a human in the loop, is where the difficulty lies. And knowing which domains even allow for this kind of automatic grading is, arguably, the more important question.

The Verification Function

If you have a function that can grade the output of your generative process as correct or incorrect without human judgment, you have an infinite supply of correctly-labelled training data. You generate a candidate via some prompting or a crafted model, you run it through the verification function, you use the result as a training signal. It is the verification function that imparts knowledge to the pipeline by acting as a grader, which knows what good looks like, rather than the blind generator.

This flow of data is precisely what modern Reinforcement Learning (RL) training pipelines are designed to exploit. Techniques like Proximal Policy Optimization (PPO) and Group Relative Policy Optimization (GRPO) allow models to improve not by memorizing more examples but by being rewarded for solutions that pass and penalized for solutions that fail [Schulman2017; ShaoEtAl2024]. Over many iterations, the model generalizes across the whole class of problems, not

because it memorized solutions, but because it has an effectively unlimited supply of freshly generated problems, each automatically verified. The verification function is what makes this possible: it grades each candidate solution, turning the output of a code generator into a labelled training example at zero marginal cost. Without it, there is no RL loop. Without an RL loop, there is no systematic improvement beyond what the pre-training data already contains.

This also means verification becomes a pacing constraint for the overall pipeline rather than generation. A good example is DeepCoder, a 14-billion-parameter coding model trained on 32 H100 GPUs, requiring over 41,000 sandboxed code executions per training step, which demanded a dedicated CPU verification service capable of running more than 100 concurrent environments [LuoEtAl2025]. The model learned faster when the verifier could keep up, not when the generator ran faster.

Why Software is Special

Software development uniquely provides a verification function of extraordinary quality. A software task has a specification — requirements, a user story, a bug report. It has an implementation — the code. And it has a test suite: a set of executable checks that determine, without ambiguity, whether the specification is implemented. Thousands of different problems can be constructed programmatically, run the tests, and get a clean binary signal back [Jimenez2024]. No taste, no interpretation, no cultural context. Just pass or fail.

This is not just a convenient property of coding tasks. It is, in a technical sense, an annotation mechanism, enabling exactly the kind of synthetic data generation pipeline that the field was searching for. You generate a problem statement programmatically, produce thousands of candidate solutions, and run the test suite on each. The passing ones become positive training examples; the failing ones become negative ones. The synthetic dataset is unlimited because the problem generator is unlimited, the labels cost nothing because the test runner is free, and the signal is clean because the tests are deterministic.

There are other factors at play with software. The sheer volume of publicly available code provides pre-training data at a scale few other domains can match. Programming languages have formal grammars, unambiguous and machine-parseable in ways that natural language is not. Code is compositional, built from discrete reusable blocks layered across levels of abstraction. Tasks decompose naturally into subtasks, each independently verifiable. These properties help, but they are not decisive**.** What matters is the presence of a reliable verification function. Without it, none of these advantages would be enough; with it, they become amplifiers.

The results are visible in benchmarks. SWE-bench, the standard evaluation for coding agents on real-world GitHub issues, has had to be repeatedly revised and hardened as agents keep saturating it [Jimenez2024]. That is not a problem with the benchmark but rather evidence that the pipeline is working at improving quality in this task. What is happening in the software engineering domain in 2026 is not evidence that the AI field is approaching general intelligence. It is, I argue, that software engineering is the domain where synthetic data and reinforcement learning are working in a tight loop to generate, verify and train. The synthetic data pipeline exploits precisely what these systems already are: extraordinarily capable generators of plausible outputs given a prompt. The verifier does the rest.

From Model to System

A base model alone, no matter how powerful, is not sufficient to achieve this level of proficiency in software engineering tasks. The coding agent harness built around it is a runtime, inference orchestration loop that equips the model with tools, tracks state and feeds the output of each step back as the input of the next. The structure echoes that of the synthetic data pipeline: constrain the process, verify the output, retry on failure. The harness works for exactly the same reason the training loop does. Every step is somehow atomic, producing a signal that can be checked, which stops errors being compounded and propagated. The current generation of tools, Claude Code and its competitors, shed the IDE scaffolding entirely. The user expresses intent in plain text; the agent researches, plans, and executes from there. The system as a whole produces something that looks like engineering. But it is, at bottom, a very sophisticated instinct machine held accountable by automated tests.

The frameworks being developed on top of coding agents lean on this approach. The Ralph pattern [Huntley2025] runs a coding agent autonomously in a loop against a product requirements document, iterating until all tasks pass their tests. Ralph does not learn between iterations. He is not intelligent, but rather persistent, and persistence inside a verifiable loop turns out to be valuable. Gastown [Yegge2026] parallelizes the process, coordinating twenty to thirty agents on the same codebase. Serial or parallel, simple or orchestrated, the architecture is the same. What the harness cannot do is care about the difference between code that passes its checks and code that is genuinely secure, efficient, maintainable, or designed for a future nobody has yet specified. While verification confirms function, this judgment remains human.

IV. The Valley of the Unverifiable

AI has not had the same impact on fields such as music, film, or literature that it has had on software engineering, and the gap is not for lack of trying. The financial incentives are enormous. The models have properties that superficially mimic creativity: temperature and sampling introduce controlled randomness, a mechanical proxy for artistic spontaneity. It is tempting to call this a matter of timing, but I think it runs deeper.

Systems like Suno and Udio can generate music in any style you care to name. Midjourney and Stable Diffusion produce striking images. Sora generates video that, in short clips, can be genuinely impressive. Since the initial wave of excitement, these tools have settled into a more modest place in the cultural landscape, useful for mockups and prototyping, but not much else. They are not producing work that people form lasting relationships with. The outputs are technically accomplished but aesthetically inert. These AI systems differ substantially under the hood, be they diffusion models, transformer hybrids or autoregressive generators, but that is not where the explanation lies. What they all share is the absence of a verification function capable of guiding improvement toward the thing that actually matters.

There is something about this aesthetic inertness that recalls the uncanny valley effect, though the discomfort operates differently here than in the original formulation. It is not that the outputs look almost-but-not-quite human. It is that they feel like the average of everything rather than the specificity of anything. Great art, even great popular art, is recognizable as coming from someone, as the product of a particular sensibility when encountering the world in a particular way. AI-generated creative content is, by construction, the synthesis of a distribution. Humans are quite sensitive to this, and the absence of a voice is striking once the initial novelty wears off.

Through the lens of synthetic data, this failure is structural, not contingent. Data generation is not the hard part. We can produce synthetic music, images, and video at scale. The problem is that we cannot tell which of it is good, and once again, the bottleneck is verification. There is no function that grades a piece of music as good or bad in the way a test suite grades code. You can generate synthetic samples for training, but labels require taste, and taste is precisely what cannot be formalized. Even human labels disagree in ways that make them a weak training signal. The RL loop has nothing to grip.

Consider a thought experiment: what if we tried to build a verification function for artistic quality? Google's MusicRL attempted exactly this, fine-tuning a music generation model using reinforcement learning on human preference signals [CideronEtAl2024]. What it produced was better alignment with average preferences, not an artistic voice. Mimicry, not music. You cannot write a test suite for the property that actually matters, which is producing work that feels like it came from someone rather than something. The gradient does not exist as a computable function, or at least, we do not know how to write it.

This feels like a genuine ceiling of the current paradigm, and not one that more compute will move. The bottleneck is not in capability, but rather the inability to formalize what good actually means in these domains. For domains where human judgment about quality is subjective, diffuse, or culturally constructed, the synthetic data pipeline that has been so productive in software engineering simply has nothing to anchor to. Science offers a parallel: AlphaFold succeeded at protein structure prediction because experimental ground truth provides a verification function. Drug efficacy does not have one. The feedback signal is a clinical trial, which takes years and cannot be generated synthetically.

Within software engineering, the verification function weakens as you move up the abstraction hierarchy, and with it so does the usefulness of synthetic data. From passing tests to good code, from good code to right architecture, from right architecture to correct product, each step is harder to verify and therefore harder to generate useful training signal from. Deciding what to build, anticipating what users will need before they know it themselves, knowing when to push back on the requirements entirely: these are the things that distinguish an experienced engineer from a code generator, and no synthetic pipeline knows how to label them yet. And the harder the judgment call, the more costly verification becomes to run at scale, until the compute required exceeds what any training pipeline can practically sustain. A few years ago, these systems could not pass a simple coding exercise, but now they ship working software. How far up the abstraction hierarchy they climb is an open question, but the direction of travel is not.

V. What the Future Tells Us

AI is advancing fast, and what seemed implausible a year ago is routine today. But competence and speed of execution are not the same as intelligence, and this distinction is worth holding onto. The word intelligence is thrown around too freely. At their core, these models are very effective generators of plausible outputs given a prompt. Provisioned with a runtime harness, they are exquisitely good at certain domain tasks, generalizing from training samples and mimicking complex chains of abstract thought. Moreover, provided their outputs can be verified and annotated, synthetic data pipelines can be built to drive further model improvement. This is a significant and economically consequential subset of cognitive labor, but it is not an intelligence signal.

Too often, the discourse tends toward an AGI-or-nothing binary that obscures more than it reveals. A more productive question, and the central one of this paper, is where else we can engineer a synthetic data pipeline grounded in a reliable verification function. Wherever we can construct such an annotation capability, synthetic data pipelines will provide data to fuel model improvement. Where we cannot, we are either looking at domains that require human judgment, or at genuinely open scientific territory. This boundary is not fixed. It can be pushed by better tooling (including harnesses), better problem decomposition, and occasionally by finding that a domain has more verifiable structure than it first appeared. This maps loosely onto a familiar distinction in computer science: problems where solutions are hard to find but easy to check are precisely the ones where automated search can make progress. Where verification itself is computationally hard or requires human judgment, the search has nothing to guide it.

Some domains look more promising than others for applying verifiable synthetic data pipelines. Formal theorem proving in mathematics may be an area where the pattern is already showing its value. Proof assistants like Lean provide a perfect verification function where the kernel accepts a proof or it does not, and systems trained on synthetic proof attempts have reached silver-medal performance at the International Mathematical Olympiad [DeepMind2025]. Formal logic and structured puzzles share the same properties [Chen2025]. The verification cost matters too: checking a mathematical proof costs under a millisecond, while code execution in a sandbox costs one to ten seconds.

Structured information retrieval follows the same logic. Whether a query returns the correct result is as verifiable as whether code passes a test. Where rules can be formalized precisely enough to encode, such as statutory law, regulatory requirements, or software access policies, the pipeline becomes tractable: synthetic propositions, transactions, documents, or agent actions can be generated and checked against the rule automatically, with no human in the loop. Legal reasoning and regulatory compliance share this structure. The wide field of cybersecurity does too, and sits even closer to software engineering. Whether a network packet matches an attack pattern, a transaction is fraudulent, a piece of code contains a known vulnerability, or whether user X is authorized by system Y to access resource Z can all be determined without human judgment, and synthetic examples of each can be

generated at scale. The verification is not as clean as a unit test, but it is far cleaner than assessing artistic quality, and the economic stakes are very high.

If this approach has any predictive value, the domains to watch are those where verifiability and economic pressure coincide. The tractability of a domain depends not just on whether a verification function exists, but on how cheap it is to run at scale. Where those conditions are met, the benchmark saturation pattern we saw in software will repeat.

Whatever lies beyond the current architectural paradigm on the road to something akin to general intelligence, my instinct is that the path runs beyond the transformer, perhaps toward something more like world models [LeCun2022]. What is certain is that whatever architectures emerge, they will need data, and the question of how to generate and verify it at scale will not go away. If anything, it becomes harder as the problems grow more ambitious and the domains less structured. Synthetic data is not a transitional workaround, it is the core challenge.

We may learn more about intelligence by asking empirically, domain by domain, where the verification function can be built and where it cannot. Any progress toward something like general intelligence will have to be grounded in good, well-verified synthetic data.

References

- Chen, Jiangjie, et al. "Enigmata: Scaling Logical Reasoning in Large Language Models with Synthetic Verifiable Puzzles." NeurIPS, 2025. arXiv:2505.19914.
- Cideron, Geoffrey, et al. "MusicRL: Aligning Music Generation to Human Preferences." ICML, 2024. arXiv:2402.04229.
- DeepMind. "Olympiad-Level Formal Mathematical Reasoning with Reinforcement Learning." Nature, 2025. https://doi.org/10.1038/s41586-025-09833-y
- Gartner. "Hype Cycle for Artificial Intelligence, 2024." Gartner Research, 2024. https://www.gartner.com/en/documents/5505695.
- Huntley, Geoffrey. "Ralph Wiggum as a 'software engineer'" 14 July 2025. https://ghuntley.com/ralph/
- Jimenez, Carlos E., et al. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" ICLR, 2024. arXiv:2310.06770.
- Kahneman, Daniel. Thinking, Fast and Slow. Farrar, Straus and Giroux, 2011.
- Karpathy, Andrej. Post on X, February 2, 2025. https://x.com/karpathy/status/1886192184808149383
- Karpathy, Andrej. Post on X, February 2, 2026. https://x.com/karpathy/status/2019137879310836075
- LeCun, Yann. "A Path Towards Autonomous Machine Intelligence." OpenReview, June 2022. https://openreview.net/forum?id=BZ5a1r-kVsf
- Lozhkov, Anton, et al. "FineWeb-Edu: the Finest Collection of Educational Content." HuggingFace, 2024. https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu
- Luo, Michael, et al. "DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level." Together AI Blog, April 2025. https://www.together.ai/blog/deepcoder

- Niklaus, Joel, et al. "The Synthetic Data Playbook: Generating Trillions of the Finest Tokens." HuggingFace, March 2026. https://huggingface.co/spaces/HuggingFaceFW/finephrase
- Schulman, John, et al. "Proximal Policy Optimization Algorithms." arXiv:1707.06347, 2017.
- Shao, Zhihong, et al. "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models." arXiv:2402.03300, 2024.
- Shumailov, Ilia, et al. "AI Models Collapse When Trained on Recursively Generated Data." Nature, vol. 631, 2024, pp. 755–759. https://doi.org/10.1038/s41586-024-07566-y
- Villalobos, Pablo, et al. "Will We Run Out of Data? Limits of LLM Scaling Based on Human-Generated Data." ICML, 2024. arXiv:2211.04325.
- Yegge, Steve. "Welcome to Gas Town." Medium, 1 January 2026. https://steve-yegge.medium.com/welcome-to-gas-town-4f25ee16dd04