

Kern: Small Code Experts Grounded in the Toolchain

2026-10-08 · @Michal

Whitepaper · version 0.2. Every number about Kern is a hypothesis to be measured; numbers from the literature are the cited authors' claims. The first measured results (milestone M1) are in the box at the end of section 10.

Abstract

Kern is a family of small (0.5-2B parameter) expert code models that run locally on a laptop and work together with a language's toolchain: the compiler, the language server (LSP) and the tests. The toolchain plays three roles. In the data it supplies diagnostics and their fixes, in training it is the reward for reinforcement learning, and during generation it constrains the model with types and closes the "compile, fix" loop.

The family has a shared trunk and swappable language modules: TypeScript/JavaScript first, then Go and Rust. A module is a LoRA adapter, a set of probes that read the model's hidden states, and a verifier adapter (the build, lint and test commands for that language).

Kern Router sits above the modules. It detects the language and the kind of task and picks the module, so the user sees one product instead of a set of models.

Our hypothesis is that Kern 1.5B reaches at least 70% of a frontier model's success rate on editing tasks (fixing compile errors, changing functions, tests, types). It should answer in under 3 s on a laptop, at zero cost per request, with a confidence signal good enough to decide between "answer, ask, decline". This document describes the architecture, the training method, the verification protocol and the benchmarks we will use to test hypotheses H1-H9.

1. Problem

Small code models are fast and local, but they invent APIs, write code that does not compile, and do not know when they are wrong. Large models are high quality, but they run in the cloud: they cost money on every request, answer in seconds, and the company's code leaves its machines.

The usual way to specialize a small model (fine-tuning, DPO on pairs generated by a large model) treats the compiler at most as a data filter. Yet the developer's machine already runs a source of truth the small model does not need to memorize: the compiler, the LSP and the tests.

The literature shows that a small model supported by such a tool matches much larger ones:

- SantaCoder 1.1B with decoding guided by static analysis compiled more often than text-davinci-003 ([Monitor-Guided Decoding](#)).

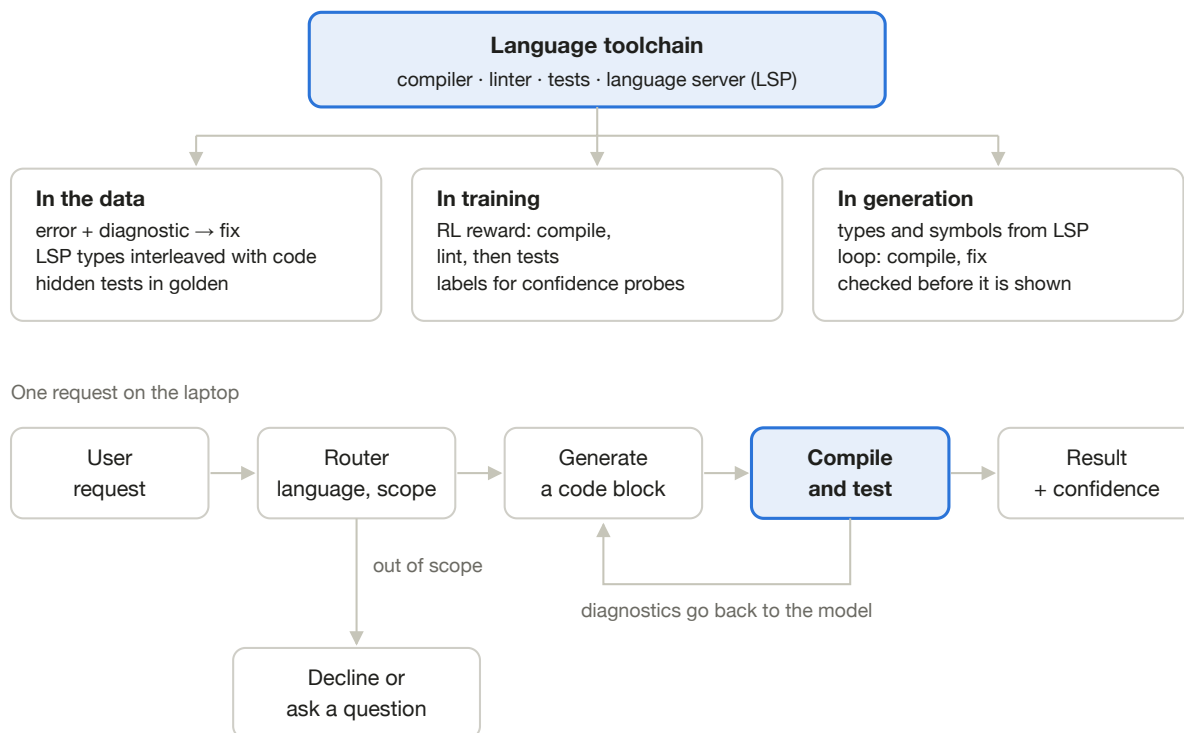
- Type-constrained decoding for TypeScript cut compile errors by more than half in 2B-34B models ([Type-Constrained Code Generation](#)).
- A 3B model that selects answers based on code execution beat GPT-4o-mini on LiveCodeBench ([S*](#)).

Kern turns this observation into a design principle: the toolchain takes part in every stage of the model's life, not only in data filtering.

2. Principle: a model coupled to the toolchain

Kern treats the toolchain as part of the model. A small model does not have to memorize every API or guess whether the code will compile, because it asks the tools that already run on the developer's machine. The principle carries over between languages, because `tsc`, `go build` and `cargo check` play the same role. It also carries beyond code to any domain with an automatic validator (accounting, construction calculations and, with limits, law), and code comes first because its validator is complete. We call models built on this principle VELMs (validated expert language models, Section 12).

The toolchain checks Kern in the data, in training and in generation



Kern principle · 3 roles of the toolchain and the flow of one request

The gate turns away out-of-scope requests, and code that fails to compile or fails its tests goes back to the model together with the diagnostics before the user sees it.

3. Architecture

Kern has four parts: a shared trunk, language modules, decision probes and a verifier adapter. There is one trunk for the whole family; the rest is a swappable module for each language.

- **Trunk.** A small multilingual code model (candidates: Qwen2.5-Coder-1.5B and Qwen3.5-2B, chosen by measurement on Kern-Golden), after continued pretraining on code interleaved with compiler diagnostics and LSP types. It stays multilingual because similar languages help each other ([Scaling Laws for Code](#)).
- **Language module.** A LoRA on all layers (including MLP) for the given language, next to a shared LoRA for tasks, routed by the file's language ([MoLE](#)). A full branch with continued pretraining is built only when LoRA is not enough, because when adding code knowledge LoRA learns less than full fine-tuning ([LoRA Learns Less and Forgets Less](#)).
- **Decision probes.** Small classifiers on hidden states: an out-of-domain gate, P(success) before generation, and a correctness estimate after generation (section 7).
- **Verifier adapter.** The build, lint, test and format commands for the language, the LSP server address, and a parser that maps diagnostics to a common format.

Kern Router: the product layer above the modules. The router detects the language and the kind of task, then picks the module, the verifier and the answer format. It works in two steps. First it checks deterministic signals: the file extension and the project manifest (`package.json`, `go.mod`, `Cargo.toml`). When that is not enough, because the request is plain natural language or the project mixes several languages, a small classifier on the trunk's hidden states decides. It returns a probability distribution over the modules plus an "out of scope" class, and asks the user when confidence is low.

Since all modules share one trunk, choosing a module means swapping a LoRA adapter in the same process: `llama.cpp` accepts an adapter with each request, and MLX loads it through `adapter_path`. The router does not load another model, so switching does not cost a reload of the weights.

Decisions about the action and the scope of the edit are special tokens in the language model head (`<act:answer|clarify|refuse|tool>`, `<fmt:block|file>`). This way they work in `llama.cpp` and MLX without a custom runtime.

Where Kern runs: Moxxy. Kern does not need its own client. It ships inside [Moxxy](#), our open-source (MIT) local agent harness, which already exists as a CLI and a desktop app for macOS, Windows and Linux (desktop v0.40.3, October 2026, developer alpha). Kern plugs in as one Moxxy model connection that bundles the trunk, the modules and the router. Compile, lint and test run as Moxxy tools behind its permission engine, which shows consequential actions before they run. Moxxy's governance layer (an identity on every action, organisation policy, audit receipts) is what the Enterprise tier builds on, and the same identity is how seats are counted. Tools other than Moxxy reach Kern through an OpenAI-compatible API priced per token.

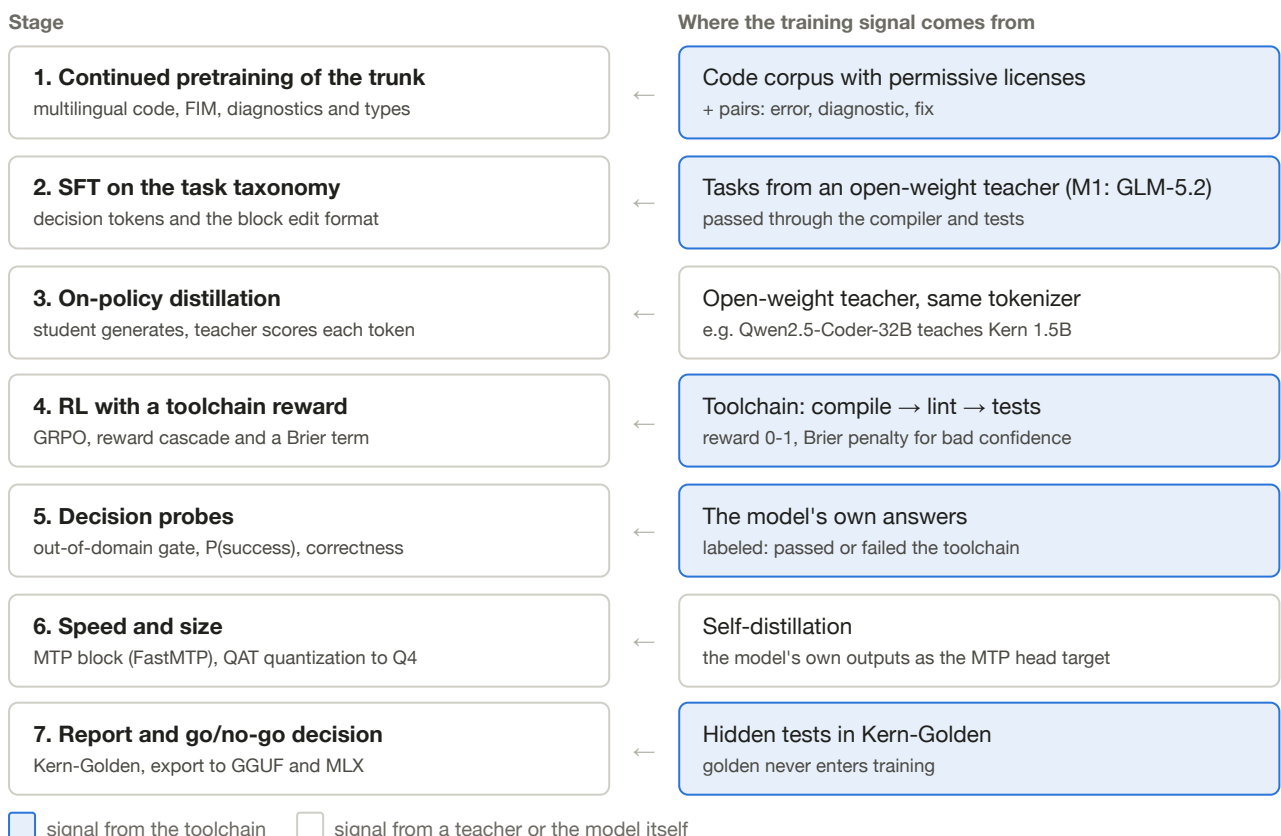
Part	Size (estimate)	Format	Trained on
Trunk	1.5B, Q4 about 1 GB	GGUF, MLX	multilingual code, diagnostics, types
Language LoRA, rank 32-64	about 75-150 MB	GGUF-LoRA, MLX adapter	code and tasks for that language
Probes (3)	under 1 MB	safetensors	the model's own answers with labels from the toolchain
Verifier adapter	a few KB	JSON manifest	needs no training

4. Training

Training has seven stages, and the toolchain supplies the signal in five of them. The order matters: RL at around 1B only works once the model, after SFT and distillation, compiles on a reasonable share of tasks.

The toolchain provides the training signal in five of the seven stages

Measured on the verify set after every stage; Kern-Golden only in the final report



Kern training pipeline · 7 stages and the signal source in each

- **Teacher for the SFT data.** In M1 the tasks for stage 2 come from GLM-5.2 (744B MoE, about 40B active, MIT license), quantized to Int4-Int8Mix and served with vLLM on 4x H200. The

toolchain accepted 58% of its edits; Qwen3-Coder-30B, which we tried first, reached 0.5% and was dropped. Throughput is about 18 batches per minute, around 77 USD per 10k records.

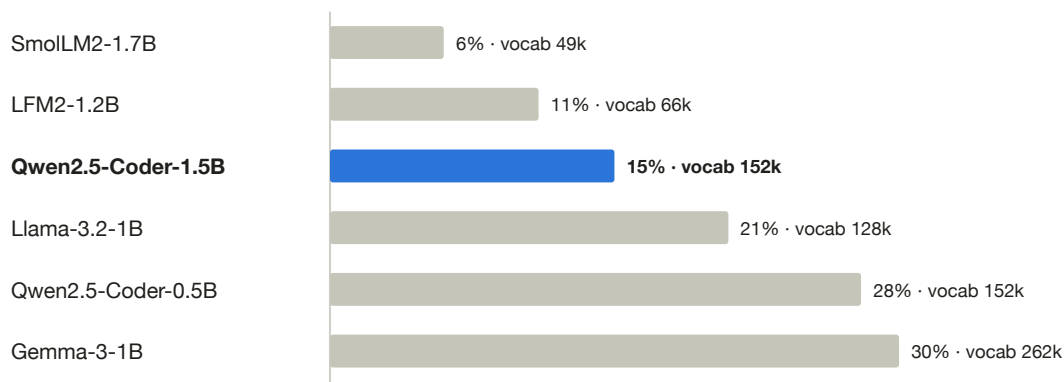
- **On-policy distillation instead of DPO on teacher pairs.** In the Qwen3 report, distillation reached 74.4% with 1,800 GPU-h, while RL alone reached 67.6% with 17,920 GPU-h ([Qwen3](#)). On code, DPO does worse than PPO ([Xu et al.](#)). In on-policy distillation the teacher scores every token the student generates, so it needs the student's tokenizer. GLM-5.2 has a different one, so we learn from it only as text, through SFT. Distillation uses a separate, smaller Qwen2.5-Coder 14B or 32B teacher. These share the student's tokenizer but have 152,064 embedding rows against the student's 151,936, and TRL's distillation trainer expects equal sizes. We therefore resize the student's embeddings to 152,064 during SFT; the extra rows are padding the tokenizer never produces.
- **RL on a reward cascade.** DeepCoder-1.5B raised HumanEval+ from 58.3 to 73.0, and VibeThinker-1.5B reached 51.1 on LiveCodeBench v6 at a training cost of \$7.8K ([VibeThinker](#)). Kern's reward ends with behavioral tests, because lint alone teaches the model to shorten its answers.
- **Short answers.** Models up to 3B learn poorly from the long reasoning chains of strong teachers ([2502.12143](#)).
- **Confidence in the reward.** A Brier term cut ECE from 0.37 to 0.03 at 7B ([RLCR](#)); it was not tested below 3B, so we retrain the probes after every RL round anyway.

5. Tokenization and output format

Kern keeps the trunk's tokenizer and shortens the output with formatting normalization and block edits. Fully replacing the tokenizer in a trained model takes more than 50B tokens of further training to recover the HumanEval and MBPP scores ([2402.01035](#)).

The vocabulary is 6-30% of a small model, 15% in the Kern trunk candidate

Share of the embedding matrix in the parameters; it is read for every generated token



model configs from Hugging Face, computed in the 2026-10 research · 6 models 0.5-1.7B

A larger vocabulary shortens the text but costs speed. Going from 64k to 128k tokens lowered decoding speed in llama.cpp by 7-10% ([Liquid AI](#)).

- **Formatting normalization.** The model learns on formatter output, generates a compact form, and Prettier, `gofmt` or `rustfmt` formats the result. In Java, C++ and C# this gave 25-36% fewer output tokens with no loss in pass@1 ([2508.13666](#)). For TSX this has to be measured, because whitespace in JSX is meaningful.
- **Appended tokens.** An optional block of a few thousand tokens per language, with embeddings initialized as the mean of their parts. The expected gain is output a few to a dozen or so percent shorter.
- **Special tokens.** Decisions (`<act:*>`), edit scope (`<fmt:*>`), FIM, edit markers and tool calls (`<lsp:hover>`, `<run:test>`).

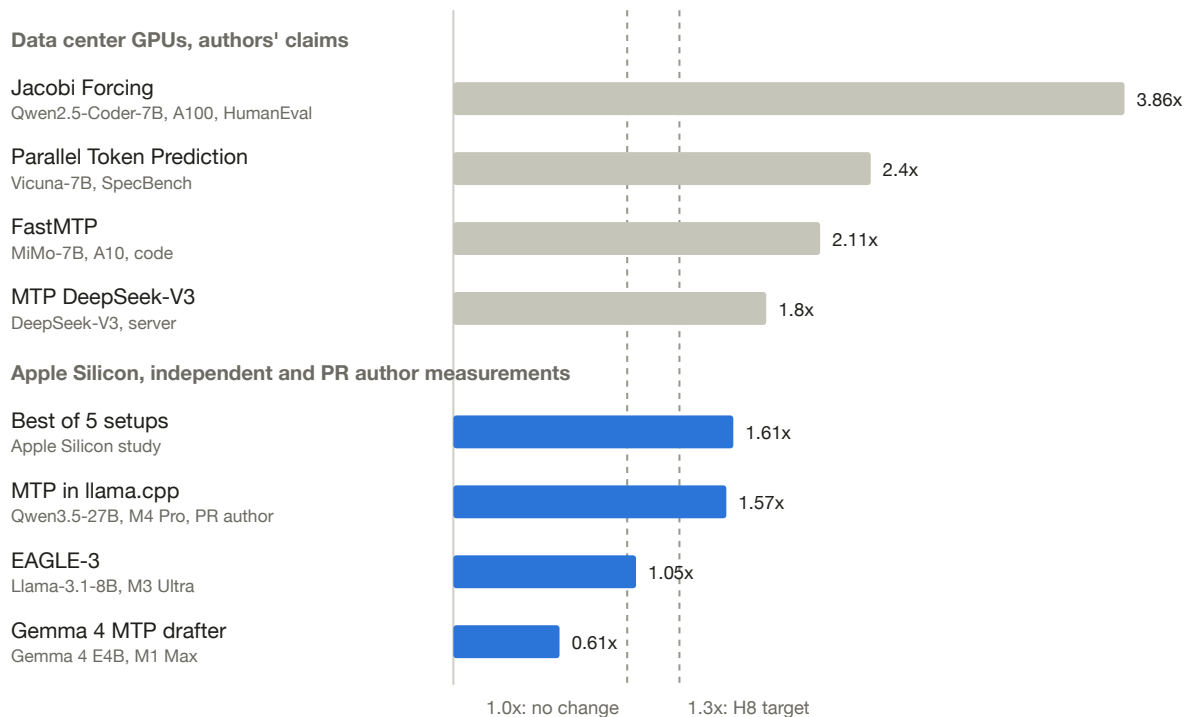
6. Generation

Kern generates changes at the level of a function or block, not whole files and not small diffs. For small models, rewriting the whole file did better than iterative search/replace blocks, and diffs won only for short, local changes ([Diffs vs. Whole Files](#)). Replacing the whole function with an adaptive choice of format matched full code at more than 30% lower latency ([To Diff or Not to Diff](#)).

- **Locator and infiller.** First the model points to where the change goes, then it writes the content. Diffusion models edit well when they know the location and lose when they have to find it themselves ([2609.38257](#)).
- **Type-constrained decoding.** For TypeScript it cut compile errors by more than half (Gemma-2-2B: from 103 to 44) at 39-52% time overhead ([2504.09246](#)). It needs a custom sampler, which is simplest in MLX.
- **Compile loop.** The compiler error goes back into the context, and the model fixes only the block it concerns.
- **Best-of-N in one batch.** Several variants are generated in parallel; the correctness probe and the compiler pick one.
- **Prompt lookup.** When editing, most tokens are copied from the input, so looking up n-grams in the context speeds things up with no training.

Speed-ups on GPUs do not carry over to the laptop: there 0.6-1.6x

How many times faster a model with a parallel token draft generates than with plain decoding



authors' claims and community measurements gathered in the 2026-10 research · 8 setups

That is why H8 measures the speed-up by wall clock on the laptop. The candidate is one MTP block trained with the FastMTP method ([2509.18362](#)); TRL has no trainer for it, so this part is our own code. llama.cpp runs MTP heads through `draft-mtp` only for architectures whose GGUF converter exports them (for example Qwen3-Next and DeepSeek), and the Qwen2.5 family is not one of them. The trunk architecture is therefore decided before phase 1: a base with MTP support, our own support in the converter and the runtime, or prompt lookup (drafts from n-grams of the context, no training) as the fallback. Methods without a separate draft are covered by the research track in section 12.

7. Decision layer and calibration

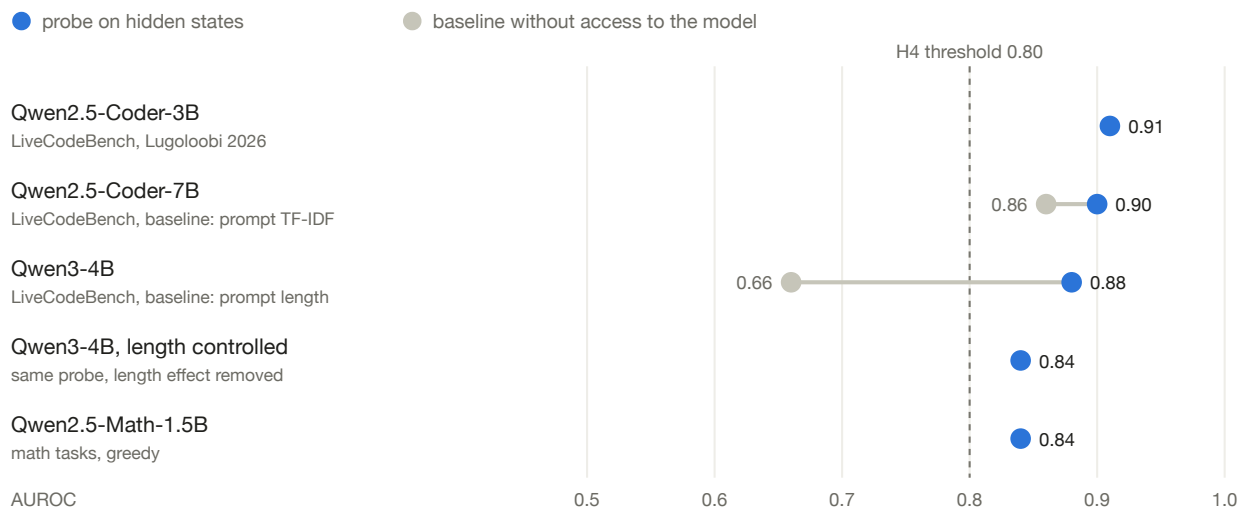
Before Kern starts writing, it needs to know three things: whether the request belongs to its module, whether it will solve it, and whether the result is correct. The model does not generate these answers as text. Small probes (logistic regression or kNN) that read the trunk's hidden states provide them. Their training labels come from the toolchain: the compiler and the tests say which of the model's answers were good.

- **Out-of-domain gate.** kNN on activations from several layers, with a bank of about 50 reference prompts, reached 99.1% F1 at detecting non-code requests ([kNNGuard](#), models from 4B). When it fires, Kern answers with a canned refusal or asks a question. The router from section 3 reads the same states.

- **P(success) before generation.** A probe on the last prompt token picks the action: answer, ask a clarifying question, best-of-N, or hand off to a larger model. Routing on such a probe cut cost by up to 70% on MATH ([Lugoloobi et al. 2026](#)).
- **Correctness after generation.** Activations contain a "correctness" direction. Reranking 10 samples along it raised HumanEval pass@1 by about 21% ([2512.07404](#), models from 7B). Kern uses it to pick a candidate before compiling. Errors on specific lines come from the compiler's diagnostics, because line-level probes are weak (top-1 0.43, [CoHalLo](#)).

Probes predict success with AUROC 0.84-0.91, but a simple baseline reaches 0.86

AUROC for predicting whether the model will solve the task, read from the hidden state before generation. 0.5 is guessing.



authors' claims: Lugoloobi et al. 2026, Di Cicco 2026 · 5 measurements, LiveCodeBench and math tasks

The chart shows the main risk: a high AUROC does not prove that the probe reads anything from the model. For Qwen2.5-Coder-7B, a TF-IDF classifier on the prompt text alone reaches 0.86, only 0.04 below the probe. So measurement in Kern follows four rules:

1. **A margin over the baseline.** H4 requires a probe AUROC of at least 0.80 and at least 0.05 above the text baseline on the same verify set.
2. **AUROC, Brier and the risk-coverage curve.** ECE alone is not enough: a model that always returns the average success rate has an ECE near zero and adds nothing. We also report pass@1 at 70, 80 and 90% coverage.
3. **Calibration after every RL round.** Plain RL with a verifiable reward makes calibration worse. RLCR, a Brier term in the reward, cut ECE from 0.37 to 0.03 at the same accuracy (Qwen2.5-7B, HotpotQA, [2507.16806](#)). Nobody has tested this below 3B. Kern adds a Brier term to the reward and retrains the probes after every round.
4. **Probes per module and per version.** On transfer to another dataset, probe AUROC fell from 0.85 to 0.72 (Lugoloobi et al.). Each module has its own probes, and the near-OOD set in golden measures this drop.

Until ECE after temperature scaling falls below 0.1, P(success) only drives decisions and is not shown to the user as a percentage. Reading a hidden state is simple in MLX, and in llama.cpp it

works through the eval callback. The cost of a probe is negligible next to prefill.

8. Kern vs. LLMs

Kern wins on cost, locality, speed and control within a narrow range of tasks, and a large model wins everything that needs broad knowledge and long reasoning. We compare it with a frontier model available through an API and with a small general code model run locally, because the latter is the real competition on a laptop.

Property	Frontier LLM (API)	Small general code model (local)	Kern
Size	hundreds of billions of parameters and more	0.5-7B	1.5B + modules
Where it runs	provider's cloud	laptop	laptop
Cost per request	per-token fee	zero	zero
Function edit latency	seconds plus network	1-5 s	target: up to 3 s (H6)
API knowledge	memorized in weights	memorized in weights	weights plus LSP during generation
Checking the result	external (agent, CI)	none	built-in compile and test loop
Confidence	uncalibrated	uncalibrated	probes on hidden states, measured by AUROC
Out-of-scope request	answers everything	answers everything, often wrongly	out-of-domain gate: decline or ask
Scope	everything	everything	one language per module, editing tasks
Code privacy	code leaves the machine	local	local
Extending	new model version from the provider	full fine-tune again	new module: LoRA, probes, verifier adapter

Out of Kern's scope: architecture design, changes across many files, explaining whole systems, and questions outside programming. The out-of-domain gate is meant to recognize such requests and pass them on.

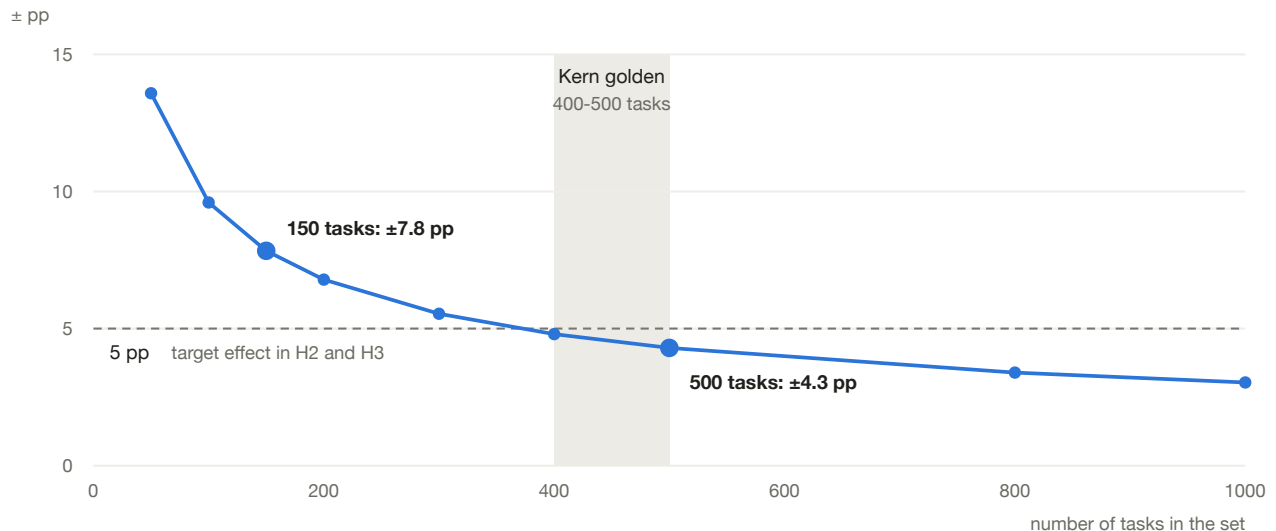
9. Verification

Every number about Kern comes from measurement on tasks the model did not see in training. The protocol has six parts.

- **Three disjoint sets.** Golden (400-500 tasks per language with hidden behavioral tests, plus a near-OOD part), verify (probe calibration, choice of checkpoint and thresholds) and train. Golden is frozen and opened only for a version report.
- **Decontamination.** N-gram and embedding deduplication between golden and all training data, including data generated by the teacher. Tasks from public repositories are taken from after the base model's cutoff date.
- **Hidden tests.** The model does not see the tests that grade it. Each test has to catch a deliberately broken version of the solution, or it is dropped from the set.
- **Paired comparisons.** Variants (with and without the toolchain, before and after RL) solve the same tasks with the same token budget. We compute the difference with a bootstrap over tasks or McNemar's test. Both variants solve most tasks the same way, so the interval for the difference is narrower than for two independent results.
- **Ablations.** Each part (toolchain data, RL, probes, MTP, type-constrained decoding) is measured by switching it off with everything else unchanged.
- **Environment record.** Every result carries the model version, quantization, hardware, toolchain version and date. Tests run in a container with no network.

An interval narrower than 5 pp needs about 370 tasks per language

Half-width of the 95% confidence interval for a single pass@1 result of 60%, in percentage points



computed from $1.96 \cdot \sqrt{p(1-p)/n}$ with $p = 0.6 \cdot 9$ set sizes

With 150 tasks and pass@1 around 60%, a result has an interval of ± 7.8 pp, so a 5 pp advantage is lost in the noise. With 500 tasks the interval is ± 4.3 pp. A hypothesis passes when the lower bound of the 95% interval for the difference clears the threshold. The mean alone is not enough.

Tests do not catch everything, so a developer reviews 50 random answers from each version (readability, fit with the project's style, unneeded changes). We report results on public benchmarks for comparison with other models. Go/no-go decisions are made on Kern-Golden.

10. Benchmarks and hypotheses

The main report rests on our own Kern-Golden. Public benchmarks serve for comparison with the literature and to check that specialization does not break general skills in the given language.

The hypotheses have thresholds proposed in advance, so that the result does not get fitted to the measurement. Each one concerns the TS module unless stated otherwise.

Benchmark	Languages relevant to Kern	What it measures	Role
Kern-Golden (own)	TS/JS, then Go and Rust	editing tasks with hidden behavioral tests plus a set just outside the domain	main report, H1-H6
Kern-CompileFix (own)	every module	fixing real compiler and linter errors	Kern-specific skill, H2
MultiPL-E	TypeScript, JavaScript, Go, Rust (out of 18)	HumanEval and MBPP translated into other languages, pass@1	comparison with published results
Aider Polyglot	JavaScript, Go, Rust (out of 6 languages)	225 hard Exercism exercises, editing in two attempts with tests	edits with a test loop
CrossCodeEval	TypeScript (next to Python, Java and C#)	code completion with context from other files in the repository	completion with context
SWE-bench Multilingual	JS/TS, Go, Rust (out of 9 languages)	300 real issues from 42 repositories	ceiling, outside the v1 goal

Hypothesis	Claim	Threshold (proposed)	Where we measure
H1: quality	Kern matches large models on narrow tasks	pass@1 at least 70% of the frontier model's score and no worse than a local, untuned Qwen2.5-Coder-7B	Kern-Golden
H2: toolchain	the compile loop and type-constrained decoding help	compile rate +15 pp, pass@1 +5 pp over the model alone	Kern-Golden, Kern-CompileFix
H3: RL	the toolchain reward adds quality	pass@1 +5 pp over SFT and distillation, at most a 2 pp drop on the control set	Kern-Golden, MultiPL-E
H4: confidence	the probes know when the model is wrong	pre-generation probe AUROC at least 0.80 and 0.05 above the text baseline; best-of-4 with reranking at least pass@1 + 8 pp	Kern-Golden
H5: out of domain	the gate recognizes requests Kern should not take	F1 at least 0.95, false refusals in the domain at most 5%	the set just outside the domain

Hypothesis	Claim	Threshold (proposed)	Where we measure
H6: latency	Kern is fast on a laptop	p50 for a function edit of up to 60 lines at most 3 s without compiling; decision at most 300 ms	MacBook with a base M chip and 16 GB, plus one Pro chip; llama.cpp and MLX; reported per chip
H7: modules	a new language as a LoRA is almost as good as full fine-tuning	Go module at least 90% of the full fine-tune's score at no more than 20% of the GPU-h	Kern-Golden-Go, MultiPL-E Go
H8: parallel draft	the MTP block speeds things up on the laptop	at least 1.3x tok/s by wall clock, pass@1 drop at most 1 pp	same laptop, same prompt set
H9: router	the router picks the module correctly and recognizes out-of-scope requests	module choice accuracy at least 98%, decision at most 50 ms beyond prefill, asks the user in at most 3% of requests	mixed set of TS, Go, Rust and out-of-scope requests

Status after M1 (2026-10-08)

M1 was a pilot, not a model release. Golden TS v0.1 has 120 tasks with hidden tests, far fewer than the 400-500 planned. On the 101 tasks that expect an answer, the untuned Qwen2.5-Coder models reach pass@1 of 0% (1.5B), 10.9% (7B) and 66.3% (32B). After SFT of the 1.5B trunk on 4.3-5.6k records from GLM-5.2, pass@1 is 33-36% and the compile rate 67-71%.

- **H2 not confirmed in this form.** Filtering the training data with the toolchain did not raise compile rate or pass@1; it only improved compliance with the Prettier format (+16 pp).
- **H1 partially.** The tuned 1.5B beats the untuned 7B baseline but stays well below the 32B.

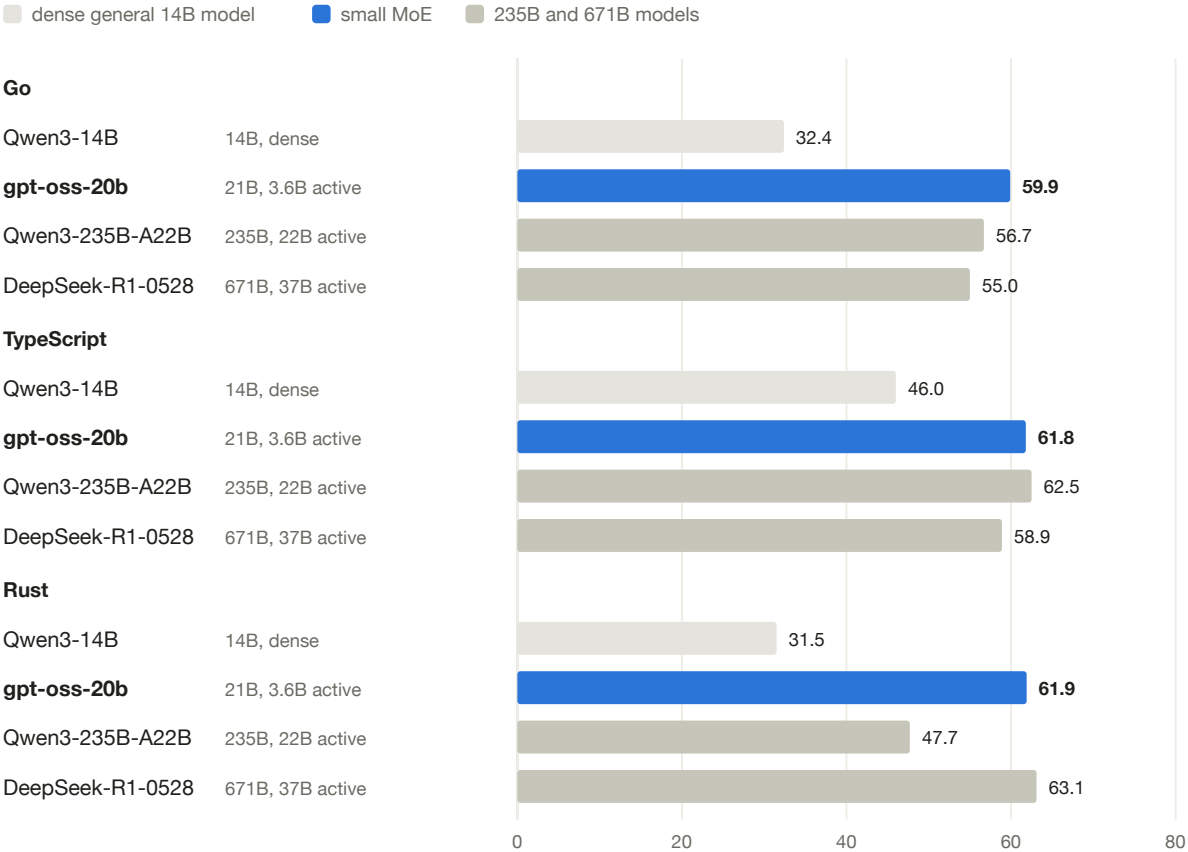
These are single runs on 101 tasks with a 3-6 pp spread between seeds, so they stay hypotheses under test. Details: [reports/m1-raport.md](#).

Projection: Kern-10B

Could Kern at about 10B parameters match 100-500B+ models within its scope, that is in one language, on editing tasks checked by tests and the compiler? Kern-10B has two variants: a dense 10B trunk or an MoE with about 3B active parameters. After Q4 quantization that is about 6 GB of weights, so it fits on a 16 GB Mac, and the MoE decodes at the speed of a 3B model. An independent multilingual benchmark shows that a small MoE already reaches the level of large models ([Multi-LCB](#)):

An MoE with 3.6B active parameters matches 235-671B models in Go, TS and Rust

Multi-LCB, pass@1 in %: LiveCodeBench tasks translated into 12 languages, models in reasoning mode



independent measurement: Multi-LCB, arXiv 2606.20517 (June 2026) · 4 models × 3 languages

This is not Kern-10B yet: gpt-oss-20b has 21B parameters in total, and the results are for competition problems solved with long reasoning. Dense general 8-14B models trail the large ones by 13 to 43 pp, depending on the language. At pass@5, an MoE with about 3B active parameters beats the pass@1 of every large model (Qwen3-30B-A3B in Rust 72.5, gpt-oss-20b in TS 78.8), so the verifier, that is the toolchain, decides parity.

Scope	What the data says	Kern-10B (projection)
Function or edit in TS with tests	MultiPL-E TS: Qwen2.5-Coder-7B 81.8, 14B 86.8, against DeepSeek-Coder-V2 236B 83.0 and GPT-4o 86.2 (different reports). Multi-LCB TS: gpt-oss-20b 61.8 against 62.5 (235B) and 58.9 (671B)	parity with 100-500B models likely
Function or edit in Go	Multi-LCB Go: gpt-oss-20b 59.9 against 56.7 (235B) and 55.0 (671B). AutoCodeBench Go: Qwen3-14B 30.4 against 31.4 (Qwen3-Coder-480B) and 38.7 (DeepSeek-R1)	parity with some of the large models likely, about 8 pp behind the best
Function or edit in Rust	AutoCodeBench Rust: Qwen3-4B 8.5 and 8B 12.6 against 36.7-38.7 (Kimi K2, DeepSeek-R1, Claude Opus 4). Multi-LCB Rust: gpt-oss-20b 61.9 against 63.1 (671B). GRPO with cargo rewards at 1.5B: compile rate from 61 to 80%, tests from 22 to 37%	the biggest risk; parity only with RL on cargo rewards and the compile loop

Scope	What the data says	Kern-10B (projection)
Fix after a compile or test error	AutoCodeBench, 3 turns of sandbox feedback: Qwen3-8B from 23.3 to 30.2, Qwen2.5-Coder-32B from 35.8 to 47.4, DeepSeek-V3 from 48.1 to 59.7. S*: Qwen2.5-Coder-7B with 16 samples and 2 debugging rounds 54.4 against 47.4 for 32B without the loop	with the loop it beats a large model answering once; with the same loop on both sides the gap remains
Multi-file edit	Aider Polyglot: Qwen3 32B 40.0% against 59.1-74.2% for large open models and 88.0% for gpt-5	out of scope, no parity
Repository-level tasks	SWE-bench: no reliable results below 20B; gpt-oss-20b 60.7 (vendor's claim) against 80.9 for Claude Opus 4.5	out of scope

Conclusion: in TS and Go, Kern-10B has a real chance of matching 100-500B models on single-language tasks with tests. In Rust it is possible but uncertain. The toolchain loop gives the advantage, but a large model with the same loop gains about as much, so at an equal compute budget the gap remains. Kern's argument then is cost and where it runs: a laptop, no per-request fees, and the code never leaves the machine.

How we will measure it: Kern-10B and 235B+ models solve Kern-Golden in the same harness, with the same attempt budget and the same compile loop on both sides. We report pass@1 without and with the loop, separately for each language, with intervals for the paired difference. Sources: [AutoCodeBench](#), [S*](#), [Aider Polyglot](#), [Rust GRPO at 1.5B](#), [Qwen2.5-Coder](#), [DeepSeek-Coder-V2](#).

11. Limitations and risks

The biggest risk is that "it compiled" gets mistaken for "it works correctly". That is why the reward and the evaluation always include hidden behavioral tests.

Risk	Effect	Mitigation
Compilation does not check what the code means	the model learns to write code that compiles but does not do what was asked; a lint-only reward made answers shorter without improving quality (Skopin, Kotelnikov 2026)	a reward cascade that ends with tests; hidden tests in Kern-Golden
RL sharpens but does not add knowledge	the base model sets the ceiling (Yue et al.)	choose the base by measurement; distillation before RL
Calibration breaks after RL	the probes start lying	retrain the probes after every RL round
The probe does not beat a simple baseline	the confidence is illusory; a TF-IDF baseline reaches AUROC 0.86 (Lugoloobi et al. 2026)	H4 requires a margin over the text baseline

Risk	Effect	Mitigation
No support in runtimes	type-constrained decoding and LSP need a custom sampler; llama.cpp runs MTP heads only for architectures that have them, and Qwen2.5 does not; MTP on a Mac gives little	MLX as the first runtime; trunk architecture chosen before phase 1; every feature behind a flag and measured by wall clock
Dependence on the base	changing the trunk forces retraining of all modules	module version pinned to the trunk version; rare base changes
Synthetic data monoculture	the model copies the style and errors of one teacher; in M1 all SFT data came from GLM-5.2	several teachers, real code, deduplication; the toolchain filter drops teacher edits that fail compile, lint or tests
Teacher tokenizer differs from the student's	GLM-5.2 cannot serve as the on-policy distillation teacher	GLM-5.2 data used as text for SFT only; a separate Qwen2.5-Coder 14B/32B teacher for distillation
Data licenses and API provider terms	legal risk	only permissively licensed code; open-weight teachers for mass generation (GLM-5.2 is MIT); every record stores its teacher, and data from a teacher with an unclear license sit in a separate version that can be dropped whole
Memory bandwidth of base M chips	decoding a 1 GB Q4 trunk is capped near 100 tokens/s at 70-120 GB/s, and a 60-line function is 500-700 tokens, so H6 can fail without speed-ups	prompt lookup, block edits instead of whole files, MTP if H8 passes; baseline latency measured on the target Macs in phase 0; H6 reported per chip
Unverified tool paths	three steps have no proven path yet: torchao int4 QAT matched to llama.cpp Q4_0 blocks, converting a PEFT LoRA into an MLX adapter, and a FastMTP trainer	each is tried on a small model in phase 0; fallbacks: Q4_K_M with an importance matrix, training the module with mlx_lm.lora or a weight-name mapping script, prompt lookup instead of MTP

12. Research directions

We test five research tracks on the same trunk as the product and compare them A/B with the base version of Kern. The first one fills a gap we did not find covered in the literature.

1. **A small model that calls the LSP.** A model below 2B trained to ask for types, definitions and references while it works. A search of arXiv found no such paper. Criterion: H1 and H2 better than with type-constrained decoding alone.
2. **Looped models.** The same layers are run several times, so the model gets more compute per byte of weights, and on a laptop memory is the limit. According to the authors, Ouro 1.4B and 2.6B match models of up to 12B ([Ouro](#)). An existing 1B model can be converted into a looped one ([Retrofitted Recurrence](#)). Criterion: better pass@1 at the same file size, with latency at most 1.5x.

3. **Block diffusion for Rust.** Diffusion models keep improving after many epochs on the same data, when autoregression has already saturated ([2507.15857](#)). A trained model can be converted cheaply ([dLLM](#)). Risk: below 1B the results are often very weak, and the runtime on a Mac is slower than autoregression.
4. **Several tokens per forward pass.** Jacobi Forcing ([2512.14681](#)) keeps an ordinary causal model and, according to the authors, gives 3.86x at 7B. Parallel Token Prediction ([2512.21323](#)) is the only method shown on code at about 1B. Criterion: H8 measured by wall clock on the laptop.
5. **LLM-JEPA with translation pairs.** An extra loss that predicts the representation of the other version of the same content, with no change to inference ([LLM-JEPA](#)). For us the pairs are the same code in TS and Go, or code and its test. Criterion: a gain on Kern-Golden-Go with the same amount of data.

Beyond code: domains with a validator

Kern's mechanism does not depend on the domain: the shared trunk, the LoRA modules, the router and the confidence probes work the same way for any task set. What makes it work in code is the validator. The compiler, linter and tests are free, automatic and unambiguous, so they supply training data, the RL reward and the run-time check. Any domain where answers can be checked automatically is a candidate for a module. Code comes first because its validator is complete. We call such a model a VELM (validated expert language model): an expert model that checks its answers with its domain's validator, such as the compiler in code or a balanced ledger in accounting. Kern is the first family of VELMs.

Domain	Validator	Strength	What is missing
Code (PoC)	compiler, linter, tests, language server	complete	nothing; this is why we start here
Accounting	double-entry balance, tax recalculation, XSD schemas for JPK and KSeF e-invoices	strong	rates and rules change every year, so current values come from retrieval
Construction	Eurocode calculations recomputed, IFC/BIM model validation and clash checks, cost estimates against unit-rate catalogues	partial	the design as a whole is still signed off by a licensed engineer
Law	the cited provision exists and was in force on the date (ISAP, EUR-Lex), deadline arithmetic, required clauses present	weak	interpretation cannot be checked automatically; the reward is partial, so the risk of reward hacking is higher; expert review is needed

Where rules run out, a generative verifier that compares the answer with a reference can supply the reward. The authors show this for medicine, chemistry, psychology and economics ([2503.23829](#)). Such a reward is soft and inherits the verifier's errors, so it does not replace a rule-based validator.

Conditions:

- **Trunk.** The current trunk is Qwen2.5-Coder, a code model. Text domains such as law or accounting in Polish need a general trunk, so a new domain means a new trunk family with its own modules, not just a new adapter. H7 is measured on programming languages and does not transfer automatically.
- **Changing knowledge.** Statutes and rates belong in a retrieval index, not in the weights of a 1.5B model. The module learns the procedure and how to use retrieval.
- **Liability.** Errors in these domains have legal and financial consequences, which makes the answer, ask or refuse probes more important than in code. On-device inference keeps privileged and client data on the user's machine.

These domains are outside the 12-month plan and budget, and we make no market-size claims for them.

References

All numeric results from these papers are their authors' claims, unless the text says otherwise.

Toolchain in decoding and verification

- [Monitor-Guided Decoding of Code LMs with Static Analysis](#)
- [Type-Constrained Code Generation](#)
- [S*: Test Time Scaling for Code Generation](#)
- [Generative Compilation \(Rust\)](#)

Training

- [On-Policy Distillation, Thinking Machines](#)
- [GKD: On-Policy Distillation of Language Models](#)
- [DeepCoder-1.5B](#)
- [VibeThinker-1.5B](#)
- [Reinforcement Learning with Calibration Rewards \(RLCR\)](#)
- [Does RL Really Incentivize Reasoning Capacity \(Yue et al.\)](#)
- [RL for small code models, Skopin and Kotelnikov 2026](#)
- [LoRA Learns Less and Forgets Less](#)
- [LoRA Without Regret, Thinking Machines](#)
- [MoLE: per-language LoRA experts](#)
- [Scaling Laws for Code](#)
- [MultiPL-T: translating data into low-resource languages](#)

Confidence and probes

- [Code Correctness Is Linearly Decodable Before Generation](#)
- [On LLMs' Internal Representation of Code Correctness](#)

- [Lugloobi et al. 2026: probes that predict success](#)
- [kNNGuard: detecting out-of-domain requests](#)
- [GenRM: Generative Verifiers](#)

Generation and speed

- [Diffs vs. Whole Files](#)
- [To Diff or Not to Diff?](#)
- [Multi-token Prediction, Gloeckle et al.](#)
- [FastMTP](#)
- [Jacobi Forcing](#)
- [Parallel Token Prediction](#)
- [Speculative decoding on Apple Silicon](#)
- [Reducing formatting tokens in code generation](#)

Research directions

- [Ouro: Looped Language Models](#)
- [Retrofitted Recurrence](#)
- [Diffusion Beats Autoregressive in Data-Constrained Settings](#)
- [dLLM: Simple Diffusion Language Modeling](#)
- [LLM-JEPA](#)
- [Crossing the Reward Bridge: Expanding RL with Verifiable Rewards Across Diverse Domains](#)

Benchmarks

- [MultiPL-E](#)
- [CrossCodeEval](#)
- [Aider Polyglot](#)
- [SWE-bench Multilingual](#)