Beyond Discrete Search: Divergent Thinking as Intention Optimization in Latent Space

Mateusz Bystroński and Grzegorz Piotrowski and Tomasz Kajdanowicz
Wrocław University of Science and Technology

Abstract

We argue that LLM-based coding agents frequently fail to solve problems that lie within the model's capacity and the bottleneck is often the conditioning context rather than the model itself. We formalize this for the full class of Turing-computable problems with verifiable specifications and introduce a framework that recasts coding as optimization over conditioning contexts that influence the generation of natural-language solution intentions. Guided by execution feedback, the method searches this continuous context space to steer a coding agent toward correct solutions. The method operates as a plug-in layer that can wrap any coding agent without modifying its architecture or weights. On SWE-Bench Verified, our method raises the resolution rate of a weak, quantized 24B open-weight model to parity with frontier models +25× its size.

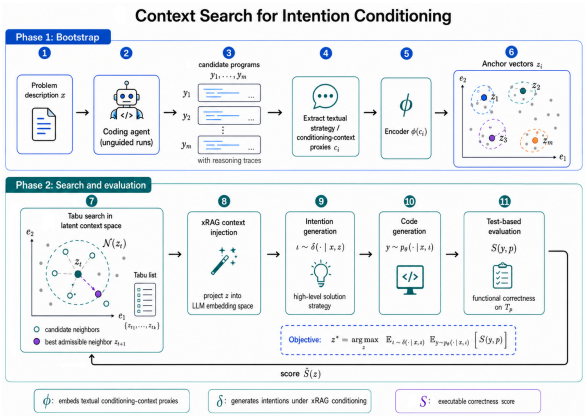

Figure 1: Workflow of the intention-guided patch generation framework. Tabu Search explores a latent intention space to iteratively refine prompts for a coding agent until a functionally correct patch.

1 Introduction

Despite rapid progress in LLM-based code generation, a persistent gap remains between what models *can* solve and what they solve on a given attempt. Practitioners routinely observe that rephrasing a prompt, supplying a hint, or describing an algorithmic strategy unlocks solutions the model previously missed. This suggests that for many problems the bottleneck is not model capacity but the conditioning context presented to it.

We make this intuition precise. We show that for every solvable programming problem there exists a context under which the language model assigns positive probability to a correct solution, and argue that coding difficulty can therefore be reframed as *context optimization*[1]. Searching over raw contexts is intractable, so we restrict the search to a structured latent space of *intentions*, natural-language descriptions of high-level solution strategies.

Our method operates in two phases entirely at inference time, treating the coding agent as a black box. A bootstrap phase runs the agent without guidance and reverse-engineers anchor intentions from its reasoning traces. A Search phase then explores the latent intention space, decoding candidate strategies, conditioning the agent on them, and feeding evaluation scores back into the search loop. No model weights are modified.

We evaluate on SWE-Bench Verified using Devstral-Small (24B, `fp8`, 100-step limit). The intention-guided framework raises the resolution rate from 45.2% to 70.2%, bringing a quantized open-weight model to parity with frontier models. Ablations show that the gains stem from structured intention exploration rather than repeated sampling: at equal compute, intention-guided search solves 35% more instances than unguided best-of-N, with strict containment of the resampling solution set.

[1]Source code is available at github.com

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop), pages 1007–1016

2 Related Work

2.1 Software Engineering Agents

LLM-based software engineering agents autonomously resolve programming tasks by reading repository code, reasoning about the problem, and producing a patch. SWE-Bench (Jimenez et al., 2024) established the standard evaluation framework: given an issue and repository, an agent must produce a patch passing held-out tests. This spurred diverse agent designs. SWE-agent (Yang et al., 2024b) introduced a structured terminal interface bridging LLM text generation and developer workflows. Agentless (Xia et al., 2025a) showed that a fixed three-stage pipeline—localise, repair, validate—can match more flexible agents. AutoCodeRover (Zhang et al., 2024) improved localisation through AST analysis.

Recent systems push resolution rates higher primarily through model scaling and task-specific training. Kimi-Dev (Yang et al., 2025) reaches 60.4% on SWE-Bench Verified via mid-training and RL with execution rewards on a 72B backbone. Live-SWE-agent (Xia et al., 2025b) achieves 77.4% by pairing frontier models with self-evolving tool creation. Across these systems, agents follow largely linear execution trajectories, committing to a solution approach early and refining through localised edits.

2.2 Search and Planning for Code Generation

Inference-time search for code generation was pioneered by AlphaCode (Li et al., 2022), which achieved competition-level performance through massive sampling and semantic clustering. CodeT (Chen et al., 2023) extended this by generating tests alongside code and retaining only mutually consistent candidates—using test agreement as an automated quality signal, an idea rooted in search-based software engineering (Le Goues et al., 2012; Fraser and Arcuri, 2011).

Tree-structured search allows branching at decision points, subsuming both paradigms. Tree of Thoughts (Yao et al., 2023) explores multiple continuations per reasoning step; LATS (Zhou et al., 2024) adds Monte Carlo rollouts for longer-horizon evaluation. In software engineering, SWE-Search (Antoniades et al., 2024) and CodeTree (Li et al., 2025b) branch over agent actions—which tool to invoke, which file to edit—but do not reconsider the high-level solution plan.

2.3 Latent-Space Steering

Alternative research explores searching over continuous vector representations rather than explicit text. While prompt/prefix-tuning (Lester et al., 2021; Li and Liang, 2021) and representation engineering (Zou et al., 2023) effectively steer models, they typically require gradient or white-box access. OPRO (Yang et al., 2024a) optimizes prompts via LLM feedback but remains in discrete space. Recently, COCONUT (Hao et al., 2025) enabled continuous latent reasoning, yet its reliance on modified training and single-turn tasks leaves a gap: whether black-box search in continuous space can navigate complex, multi-step repository-level coding.

2.4 Inference-Time Compute Scaling

Rather than training larger models, one can invest more compute at inference time. (Snell et al., 2024) showed this trade-off can be surprisingly favourable: a well-chosen test-time strategy can match a model $14\times$ larger on moderately difficult problems. The simplest approach—repeated sampling—yields log-linear coverage growth (Brown et al., 2024); CodeMonkeys (Ehrlich et al., 2025) applied this to SWE-Bench, reaching 57.4%. More sophisticated methods guide the search: (Li et al., 2025a) showed that mixing parallel and sequential sampling pushes a 3B model past GPT-4o-mini, while (Ma et al., 2025) achieved competitive SE performance from a 32B model with reward-guided search.

3 Method

3.1 Coding as Context Optimization

Let $\mathcal{P}$ denote a family of solvable programming problems, where

$$\mathcal{P} \subseteq \{f \mid f \text{ is Turing-computable}\}$$

Let each problem $p \in \mathcal{P}$ be specified by a natural-language description x and a test specification $\mathcal{T}_p$ of input-output pairs. A program y is correct iff $\forall (i, o) \in \mathcal{T}_p,\ y(i) = o$. Let $p_\theta(y \mid c)$ denote a language model generating y conditioned on context c.

We call language model p_θ *ideal* if it assigns positive probability to every linguistically valid response. Under this assumption, for every solvable problem there exists a context $c^\star$ (e.g., the problem description concatenated with a correct solution and an instruction to reproduce it) such

that $p_\theta(y^\star \mid c^\star) > 0$ for some correct $y^\star$. The proof is immediate: verbatim reproduction of in-context code is a valid linguistic behaviour.

Since modern LLMs are trained to model language over massive and highly diverse corpora, including code and problem-solution text, we treat them as practical approximations of this idealized language model. Under this assumption, coding can be reframed as search over contexts that maximize the probability of correctness:

$$c^\star = \arg\max_c \ \mathbb{P}_{y\sim p_\theta(\cdot|c)}\big[y \in \mathcal{Y}(p)\big],$$

where $\mathcal{Y}(p)$ is the set of all correct programs for p. Since this probability is inaccessible, we approximate it via single-sample evaluation: draw $y \sim p_\theta(\cdot \mid c)$ and score it against $\mathcal{T}_p$ (context with higher success probability necessarily yields a correct first sample with higher probability).

3.2 Context Search for Intention Conditioning

Searching directly over programs is difficult. We therefore introduce an intermediate generation step: before writing code, the agent first generates a natural-language intention, i.e. a high-level solution strategy. Our search optimizes this process indirectly: instead of editing the intention text itself, we optimize the conditioning signal under which the intention is generated.

Let $\mathcal{I}$ denote the space of natural-language intentions. For a problem description x, the model first samples an intention

$$\iota \sim p_\theta(\cdot \mid x, c),$$

where c is an additional conditioning context, and then generates a program conditioned on the problem and the sampled intention:

$$y \sim p_\theta(\cdot \mid x, \iota).$$

Thus, the searched object is not a program and not the final intention text, but a context that changes the distribution over intentions.

Prior work on continuous semantic conditioning (Bystroński et al., 2026) motivates this formulation: additional context, even when not explicitly task-specific, can coherently shift the model's generation distribution. Searching over raw textual contexts is intractable, so we search over their continuous representations. Let

$$z \in \mathbb{R}^d$$

denote a latent conditioning vector. In our implementation, z is passed through the learned xRAG projector (Cheng et al., 2024) and injected into the LLM input-embedding space. This produces an effective conditioning signal for intention generation.

We denote the resulting induced distribution over intentions by

$$\delta(\iota \mid x, z) = p_\theta(\iota \mid x, z_{\mathrm{xRAG}}),$$

where z_{xRAG} is the projected representation injected into the model. The vector z therefore does not represent the intention directly; it defines the conditioning signal under which the intention is sampled.

The objective is to find a conditioning vector that maximizes expected functional correctness:

$$z^\star = \arg\max_{z\in\mathbb{R}^d} \ \mathbb{E}_{\iota\sim\delta(\cdot|x,z)}\mathbb{E}_{y\sim p_\theta(\cdot|x,\iota)}\big[S(y,p)\big].$$

The score S is computed by executing the generated program against the test specification. In our SWE-Bench setup we use

$$S(y,p) \ = \ r_{F2P}(y,p) \ + \ \tfrac{1}{60}\, r_{P2P}(y,p),$$

where r_{F2P} is the fraction of failing-to-passing tests that pass under y, and r_{P2P} is the fraction of passing-to-passing tests that remain green. A candidate is resolved iff $r_{F2P} = r_{P2P} = 1$.

Because the expectation is inaccessible, we estimate each latent point by a single execution.

3.3 Bootstrap and Search

Phase 1: Bootstrap. The agent executes repeatedly without intention guidance, producing candidates $\{y_1, \ldots, y_m\}$. From each candidate and its reasoning trace, an anchor intention ι_i is extracted and embedded as $z_i = \phi(\iota_i)$, initializing the search region in $\mathcal{Z}$.

Phase 2: Tabu Search (Glover, 1986). At iteration t, a neighborhood operator $\mathcal{N}(z_t)$ proposes candidate vectors near z_t. Each candidate is decoded into an intention, executed by the agent, and scored. The next state is:

$$z_{t+1} = \arg \max_{z\in\mathcal{N}(z_t)\setminus\mathcal{T}_t} \widehat{S}(z),$$

where $\mathcal{T}_t$ is a tabu set of recently visited intentions that prevents cycling and encourages exploration of new strategies.

4 Experiments

This section presents proof-of-concept experiments establishing the viability of latent intention search as an axis of inference-time compute scaling; comprehensive comparison across models, benchmarks, and search hyperparameters is left to future work.

4.1 Setup

We evaluate on **SWE-Bench Verified** (bash-only track, dev split, n=500) (Jimenez et al., 2024), a curated set of real-world GitHub issues from major Python repositories. All agents interact with the repository through a minimal ReAct loop (*mini-SWE-agent*) with access only to a bash shell. A patch is correct iff all associated tests pass in a containerized environment.

The base model p_θ is **Devstral-Small-2507** (24B), served via vLLM with `fp8` quantization and a 64k-token context window at temperature τ=0. The CSC module (ϕ, δ) runs separately at 4-bit precision. The bootstrap phase executes the unguided agent with up to M=100 steps per attempt 3 times; the Tabu Search phase proceeds for up to R=50 rounds with the same per-round step budget. The neighborhood operator uses local perturbation scale σ_{local}=300 and kick scale σ_{kick}=1,000. The tabu tenure is τ_{tabu}=50, with stagnation threshold s=5 and kick probability p_{kick}=0.25.

4.2 Baselines

We compare against two categories of baselines:

Same-model references. (i) *Devstral-Small (official)*: the unquantized model with the standard 250-step mini-SWE-agent scaffold (53.6%). (ii) *Devstral-Small (degraded)*: our `fp8`-quantized, 100-step setup without intention guidance (45.2%), serving as the direct ablation baseline.

Frontier models. Scores for Gemini 3 Pro, DeepSeek V3.2 (685B), Claude Sonnet 4.5, and GPT-5.2 are taken from the official SWE-Bench leaderboard, all evaluated with the same mini-SWE-agent scaffold.

4.3 Results

Table 1 presents the main results. The degraded baseline resolves 45.2% of instances. Our method raises this to **70.2%** (+25.0 pp), entirely through search-time optimization over $\mathcal{Z}$—without modifying model weights. Of the 274 instances unsolved after bootstrapping, the search phase recovers 125 (45.6% conversion rate). This brings a relatively old, quantized 24B model above Gemini 3 Pro and on par with DeepSeek V3.2 (685B), narrowing the gap to Claude Sonnet 4.5 and GPT-5.2 to within 1–3 pp.

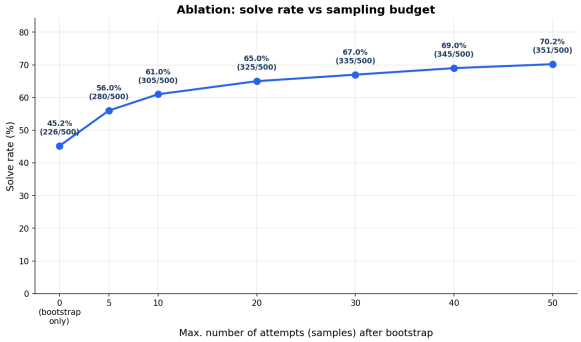

Figure 2: Resolution rate as a function of the Tabu Search round budget R.

4.4 Ablation: Search Budget

Figure 2 shows how the resolution rate scales with the number of Tabu Search rounds R. The first 5 rounds yield the steepest gains, lifting the solve rate from 45.2% to 56.0% (+10.8 pp, 54 additional instances). Subsequent rounds continue to contribute, though with diminishing returns: rounds 5–20 add a further 9.0 pp (45 instances), while the final 30 rounds ($R = 20 \rightarrow 50$) contribute 5.2 pp (26 instances).

4.5 Ablation: Intention Search vs. Naive Resampling

To isolate the effect of intention-guided strategy-level search from repeated execution, we compare both methods on a stratified sample of 50 bootstrap-unsolved instances with an equivalent budget of R=50 agent executions per instance. Naive resampling (best-of-N) solves 17/50 (34.0%); intention search solves 23/50 (**46.0%**), a 35% relative improvement (Table 2). Instance-level analysis reveals a strict containment: every instance solved by resampling is also solved by intention search; the converse does not hold.

5 Discussion

Latent intention search introduces a new paradigm for LLM-based problem solving: instead of improving the model, search over the space of strategies that condition it. The formal framework makes no assumptions specific to software engineering, the context optimization approach holds for any

Table 1: SWE-Bench Verified (bash-only, dev, n=500). Same-model baselines share our base model; frontier scores use the standard mini-SWE-agent scaffold from the official leaderboard. No model weights are modified by our method.

Model	**Method**	**Params**	**Res. (%)**
Same-model baselines			
Devstral-Small	mini-SWE-agent (official)	24B	53.6
Devstral-Small	mini-SWE-agent (limited)	24B	45.2
Ours			
Devstral-Small	Intention search + mini-SWE-agent (limited)	**24B + 14B**	**70.2**
Frontier models (leaderboard)			
Gemini 3 Pro	mini-SWE-agent	—	69.6
DeepSeek V3.2	mini-SWE-agent	685B	70.0
Claude Sonnet 4.5	mini-SWE-agent	—	71.4
GPT-5.2	mini-SWE-agent	—	72.8

Table 2: Intention search vs. naive resampling on 50 bootstrap-unsolved instances (R=50 executions each).

Method	**Resolved (%)**
Naive resampling (best-of-N)	34.0
Intention search (ours)	**46.0**

problem $p \in \mathcal{P}$, i.e., for the full class of Turing-computable functions with verifiable specifications. Any domain where a candidate solution can be automatically evaluated—theorem proving, hardware synthesis, constrained optimization— admits the same search over a latent strategy space.

The key structural requirement is a scoring function $S(y, p)$ that provides a signal for the search loop. In SWE-Bench this is extractable from tests pass or fail; richer signals such as partial test scores, proof-checking, or simulation-based evaluation could enable more efficient search in other domains.

Despite the limitations discussed below, the results are strong: a weak quantized 24B model reaches parity with frontier models. This suggests that the paradigm has substantial headroom, and that systematic improvements to the search method, agent integration, and cross-domain evaluation may yield further gains.

Limitations

A notable gap in our evaluation is the absence of controlled comparison with other search-based methods. This reflects the novelty of the setting itself: to our knowledge, no prior work on the SWE-Bench Verified leaderboard attempts to elevate a small open-weight model to frontier-level performance through inference-time search alone. The leaderboard entries rely on increasingly capable base models.

Our evaluation is limited to a single base model (Devstral-Small, 24B) and one benchmark (SWE-Bench Verified). While the formal framework applies to any LLM, we have not yet empirically verified that the gains transfer to other base models or model scales. We consider the present results sufficient to establish the viability of the paradigm; this paper is an invitation to further—necessarily costly—experimentation across models and domains.

The comparison with frontier models is asymmetric: their scores reflect single-pass inference with a 250-step budget, while our method invests up to R=50 search rounds per instance. This is by design—we argue that inference-time search is a legitimate and underexplored axis of scaling—but a complete comparison would apply the same framework on top of frontier models, which we leave to future work due to cost constraints.

A practical challenge of benchmarking in a fast-moving field is that infrastructure evolves during experimentation. There is a version mismatch in the evaluation scaffold: Devstral-Small and our method were evaluated using mini-SWE-agent v1.17, while the frontier model scores in Table 1 are from v2.0.0, which was released during the course of our experiments. The newer version improves baseline scores (e.g., GPT-5.2 scored 69.0%

under v1.17 vs. 72.8% under v2.0.0). We consider comparing against frontier models in their stronger configuration to be the appropriate choice, as it avoids overstating our relative gains.

Several axes of improvement remain unexplored. The search algorithm is not yet optimized: the bootstrap phase was run only 3 times, and the Tabu Search hyperparameters (σ_{local}, σ_{kick}, τ_{tabu}, s, p_{kick}) were tuned heuristically with a strong bias toward exploration over exploitation. Rebalancing this trade-off, for instance, by annealing the perturbation scale or reducing the kick probability as search progresses could allow the agent to more thoroughly exploit promising regions of the intention space and is left to future work. Alternative metaheuristics such as genetic algorithms or CMA-ES may navigate the intention space more effectively. The current architecture treats the coding agent as a black box: the intention search operates as an outer loop around the agent's own ReAct loop, with no communication between the two. This decoupled design was a deliberate choice—it allows us to evaluate intention search in isolation using a standardized agent scaffold, ensuring that gains are attributable to the search over $\mathcal{Z}$ rather than to architectural modifications. However, tighter integration—e.g., feeding intermediate agent signals back into the intention search, or conditioning individual agent steps on sub-intentions—could improve both efficiency and effectiveness. This work is a proof of concept for latent intention search; systematic investigation of the search method, its hyperparameters, and agent integration is left to future work.

The Tabu Search phase requires repeated full agent executions, making it substantially more expensive than single-pass inference. Each search round launches the complete ReAct loop (up to 100 steps of repository interaction), and the cost scales linearly with the round budget R. We observe diminishing returns beyond $R \approx 20$ rounds (Figure 2), suggesting that adaptive termination could reduce cost, but we have not yet explored this. A promising direction is to replace full agent executions with lightweight rollouts from a code world model (FAIR CodeGen team et al., 2025), simulating the outcome of a coding strategy without actually running it, potentially reducing the cost per search iteration by orders of magnitude.

Finally, the CSC module (ϕ, δ) introduces an additional dependency: the quality of the latent space determines the expressiveness of the search. A poorly structured intention space could limit the framework's ability to discover novel strategies beyond those seen during bootstrapping. The module also adds 14B parameters (encoder and decoder combined), which, while frozen during search, increase the total inference-time memory footprint beyond the 24B base model alone. These constraints are not inherent to the paradigm—a lighter-weight projector or a differently trained latent space could reduce both the memory overhead and the dependency on CSC's specific inductive biases.

Acknowledgments

This work was supported by the AITAX (AI Tax Advisor) project under the grant FENG.02.02-IP.05-0314/23, Action 2.2 FIRST TEAM, European Funds for a Modern Economy Programme 2021–2027 (FENG). Calculations have been carried out in the Wroclaw Centre for Networking and Supercomputing (http://www.wcss.pl) as well as using services of CLARIN-PL.

References

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-search: Enhancing software agents with Monte Carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*.

Mateusz Bystroński, Doheon Han, Nitesh V. Chawla, and Tomasz Kajdanowicz. 2026. Geometry of knowledge allows extending diversity boundaries of large language models. *Preprint*, arXiv:2507.13874.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code generation with generated tests. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. 2024. xrag: Extreme context compression for retrieval-augmented generation with one token. *Preprint*, arXiv:2405.13792.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling test-time compute for software engineering. *arXiv preprint arXiv:2501.14723*.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, and 1 others. 2025. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*.

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, pages 416–419.

Fred Glover. 1986. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2025. Training large language models to reason in a continuous latent space. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can language models resolve real-world GitHub issues? In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Dacheng Li and 1 others. 2025a. S*: Test-time scaling for code generation. *arXiv preprint arXiv:2502.14382*.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2025b. CodeTree: Agent-guided tree search for code generation with large language models. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the Association for Computational Linguistics (ACL-IJCNLP)*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Yingwei Ma and 1 others. 2025. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. *arXiv preprint arXiv:2503.23803*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025a. Agentless: Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE).

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025b. Live-SWE-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024a. Large language models as optimizers. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Zonghan Yang, Shengjie Wang, Kelin Fu, and 1 others. 2025. Kimi-dev: Building a more practical AI software engineer. *arXiv preprint arXiv:2509.23045*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous program improvement. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language agent tree search unifies reasoning, acting, and planning in language models. In *Proceedings of the International Conference on Machine Learning (ICML)*.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, and 1 others. 2023. Representation engineering: A top-down approach to AI transparency. *arXiv preprint arXiv:2310.01405*.

A Case Study

To illustrate the behavior of intention search, we analyze a concrete instance from `SWE-Bench Verified`: `sympy__sympy-20801`. The underlying bug arises from an inconsistency in SymPy's comparison semantics between symbolic booleans and numeric values. In particular, the expression

```
S(0.0) == S.false
```

incorrectly evaluates to `False`. The objective of the search is therefore to identify a modification that reconciles this semantic mismatch while preserving the behavior of the existing test suite.

Table 3 shows the full trajectory of the intention search. Each round proposes a high-level repair hypothesis ("intention") that the coding agent attempts to implement. The score reflects the balance between fail-to-pass and pass-to-pass test ratios. Scores greater than 1.0 indicate that the generated patch resolves the failing tests without introducing regressions.

Over the course of 24 rounds, the search explores a wide variety of repair hypotheses before converging on a resolving intention at round 23. Although the trajectory appears diverse at the surface level, many intentions correspond to variations of a smaller number of *repair strategy families*. These include:

- **Representation redefinition**, which alters the internal representation of objects (e.g., redefining `S.false` as a numeric zero).
- **Equality semantic modification**, which changes the behavior of equality itself (e.g., overriding `__eq__` or adjusting comparison logic).
- **Canonicalization before comparison**, where both operands are mapped into a shared representation prior to equality testing.
- **Alternative API strategies**, which introduce new operators or interfaces for equivalence (e.g., a dedicated `.equiv()` method).
- **Type-system restructuring**, which attempts to resolve the issue through class hierarchy changes.
- **Numeric tolerance heuristics**, which treat the mismatch as a floating-point comparison problem.
- **Prohibition strategies**, which reject the comparison as ill-defined.
- **Specification reinterpretation**, which modifies the comparison target or expected semantics rather than the underlying mechanism.

Table 4 summarizes the strategy families encountered during the trajectory.

Table 3: Intention search trajectory for `sympy__sympy-20801`. Each round proposes a high-level repair hypothesis that the coding agent attempts to implement.

Round	Res.	Strategy Family	Representative Intention
0		Representation redefinition	Redefine `S.false` as a numeric zero representation.
1		Spec reinterpretation	Replace comparison target with canonical `S.Zero`.
2		Alternative API	Introduce a new equality operator avoiding mixed-type comparison.
3		Spec reinterpretation	Replace values with explicit `S.True`/`S.False`.
4		Prohibition strategy	Reject the comparison as undefined.
5		Type-system reasoning	Resolve inconsistency through class hierarchy semantics.
6		Equality modification	Change comparison logic so the expression evaluates to `True`.
7		Equality modification	Treat numeric zero and symbolic false as equivalent in equality.
8		Numeric heuristic	Use tolerance-based numeric comparison.
9		Alternative API	Introduce a dedicated `.equiv()` method.
10		Equality modification	Implement boolean equivalence semantics.
11		Equality modification	Apply mathematical logic equivalence between zero and false.
12		Type-system restructuring	Introduce a new `BooleanType` class.
13		Equality modification	Define asymmetric equality semantics.
14		Type-system restructuring	Enforce uniform boolean comparison rules.
15		Equality modification	Treat symbolic false as numeric zero during equality checks.
16		Canonicalization	Introduce helper conversion for numeric equivalence.
17		Equality modification	Override equality operator implementation.
18		Representation redefinition	Redefine the internal representation of `False`.
19		Representation redefinition	Reassign `S.false` to a numeric symbol.
20		Numeric heuristic	Use `isclose()` for floating-point comparison.
21		Numeric heuristic	Combine tolerance comparison with equality override.
22		Canonicalization	Convert symbolic values to numeric equivalents before comparison.
23	✓	**Canonicalization**	**Map symbolic booleans and numeric values to floats before equality testing.**

Table 4: Generalized repair strategy families explored during the intention search for `sympy-20801`. Although the trajectory contains 24 rounds, these correspond to a smaller set of reusable repair hypotheses about how the bug should be fixed.

#	Strategy Family	Rounds	Best Score
1	Representation redefinition (modify internal representation of objects)	0,18,19	0.017
2	Spec reinterpretation / target substitution	1,3	0.017
3	Alternative API or operator introduction	2,9	0.017
4	Prohibition strategy (reject comparison as ill-defined)	4	0.017
5	Type-system restructuring	5,12,14	0.017
6	Equality semantic modification	6,7,10,11,13,15,17	0.017
7	Numeric tolerance heuristics	8,20,21	0.017
8	Canonicalization before comparison	16,22	0.016
9	**Canonicalization via float conversion (winner)**	**23**	**1.017**