University of Virginia

Section 8.1: Testing Time Scaling

2026 Spring

LLM Agents Foundation & Applications

Student Team / 20260407

Roadmap

- S1: LLM Basic Alignment
- S2: LLM Alignments for Reasoning
- S3: Agent applications
- S4: LLM Data synthesis
- S5: Agent Memory
- S6: LLM model serving
- S7: Agent Evaluation and Attack/Defense Landscape
- S8: Agent Planning / Testing Time Scaling → This lecture!
- S9: World Modeling for GenAI Agents
- S10: Multi-Agents

[Submitted on 31 Mar 2025 (v1), last revised 4 May 2025 (this version, v3)]

A Survey on Test-Time Scaling in Large Language Models: What, How, Where, and How Well?

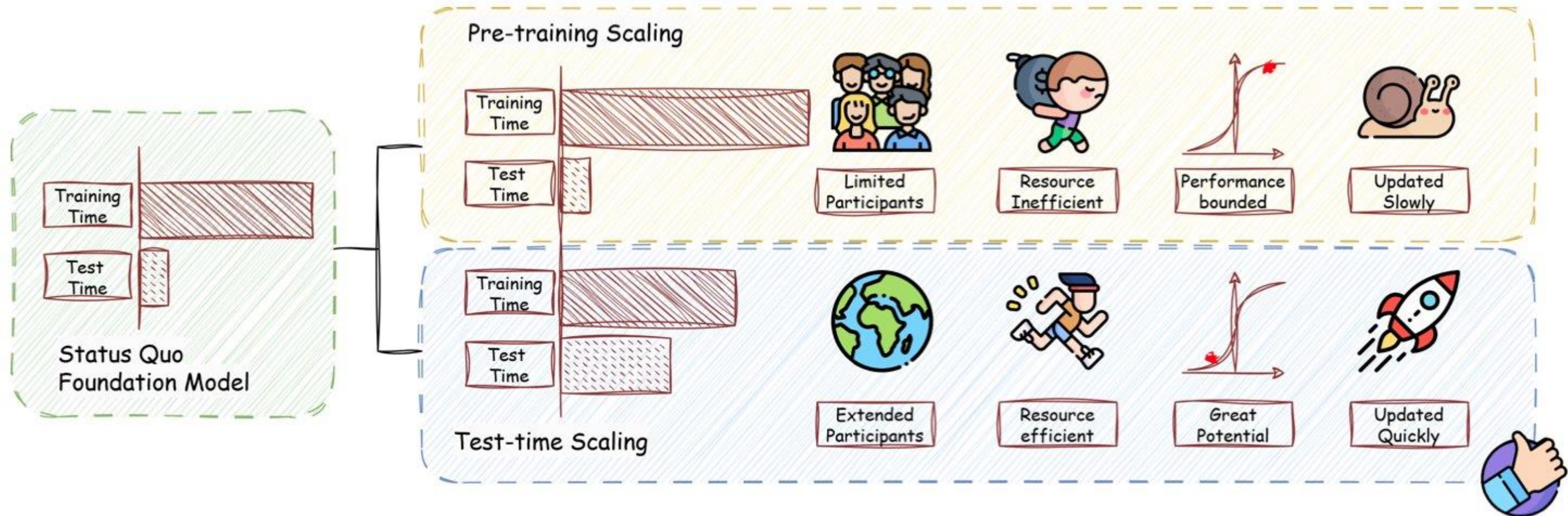
Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, Chen Ma

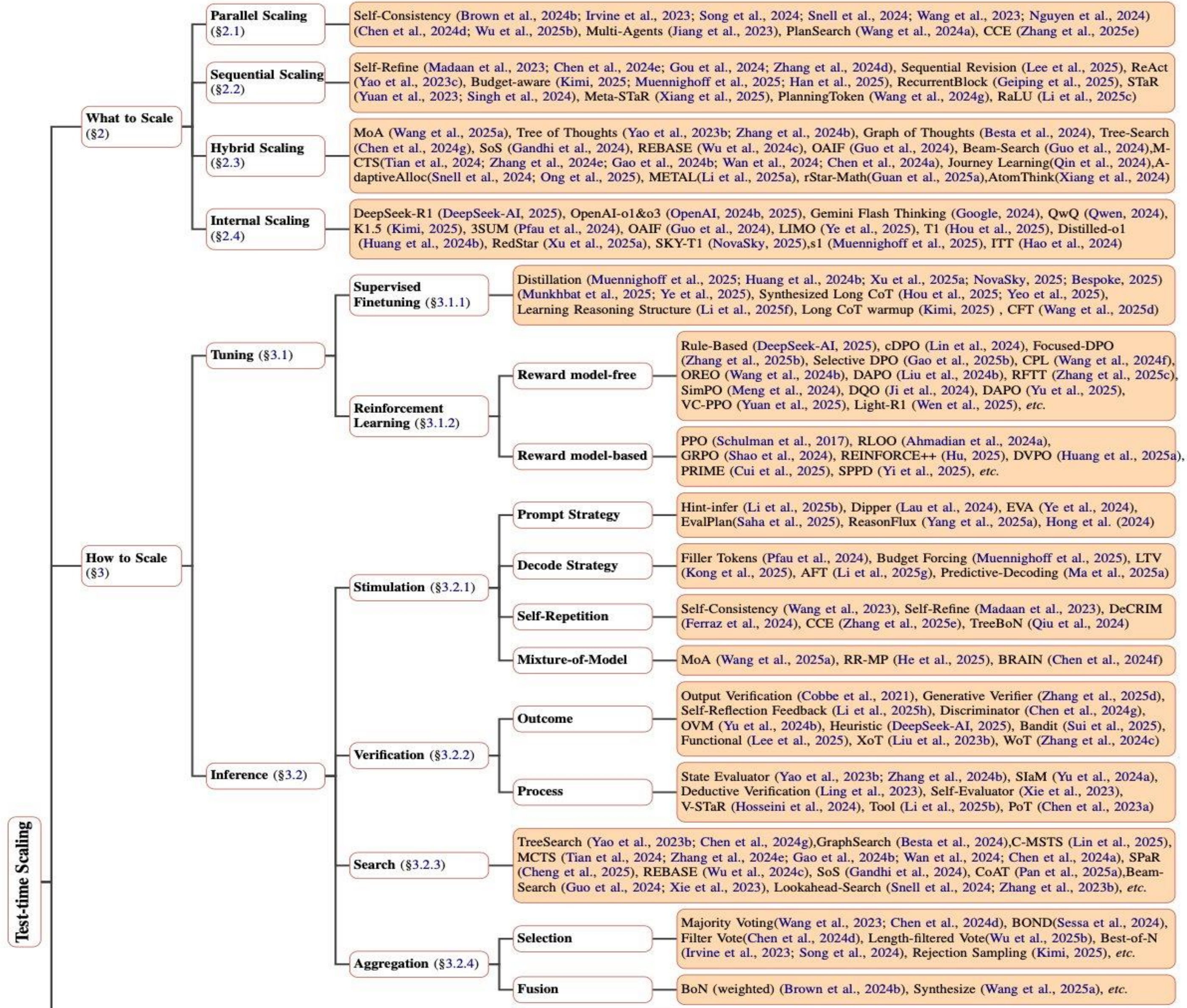
As enthusiasm for scaling computation (data and parameters) in the pretraining era gradually diminished, test-time scaling (TTS), also referred to as ``test-time computing" has emerged as a prominent research focus. Recent studies demonstrate that TTS can further elicit the problem-solving capabilities of large language models (LLMs), enabling significant breakthroughs not only in specialized reasoning tasks, such as mathematics and coding, but also in general tasks like open-ended Q&A. However, despite the explosion of recent efforts in this area, there remains an urgent need for a comprehensive survey offering a systemic understanding. To fill this gap, we propose a unified, multidimensional framework structured along four core dimensions of TTS research: what to scale, how to scale, where to scale, and how well to scale. Building upon this taxonomy, we conduct an extensive review of methods, application scenarios, and assessment aspects, and present an organized decomposition that highlights the unique functional roles of individual techniques within the broader TTS landscape. From this analysis, we distill the major developmental trajectories of TTS to date and offer hands-on guidelines for practical deployment. Furthermore, we identify several open challenges and offer insights into promising future directions, including further scaling, clarifying the functional essence of techniques, generalizing to more tasks, and more attributions. Our repository is available on this https URL

Test-time scaling emerges as prominent research focus

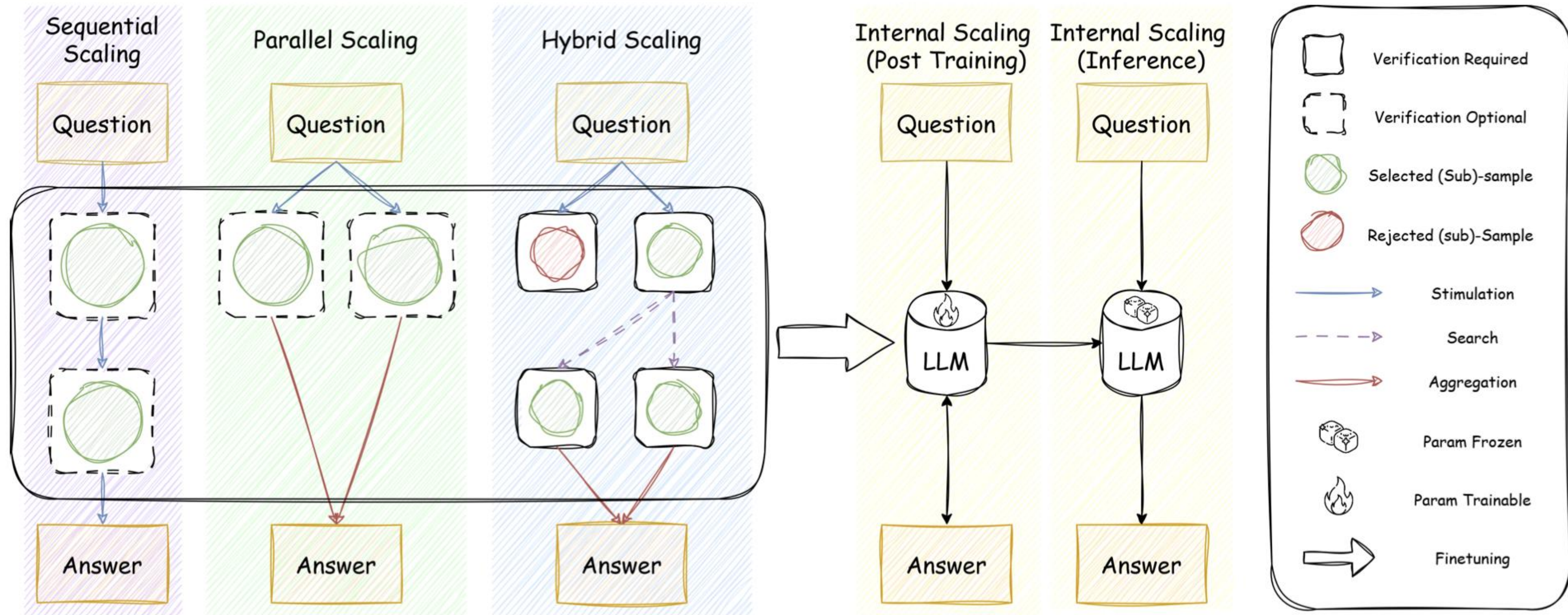
- What is Test Time Scaling?
  - Allocate additional resources during inference (kind of like humans)
- Enabling breakthroughs in specialized and general tasks
  - OpenAI o1, DeepSeek's R1
- Need for comprehensive survey for systemic understanding
  - What to scale
  - How to scale
  - Where to scale
  - How well to scale

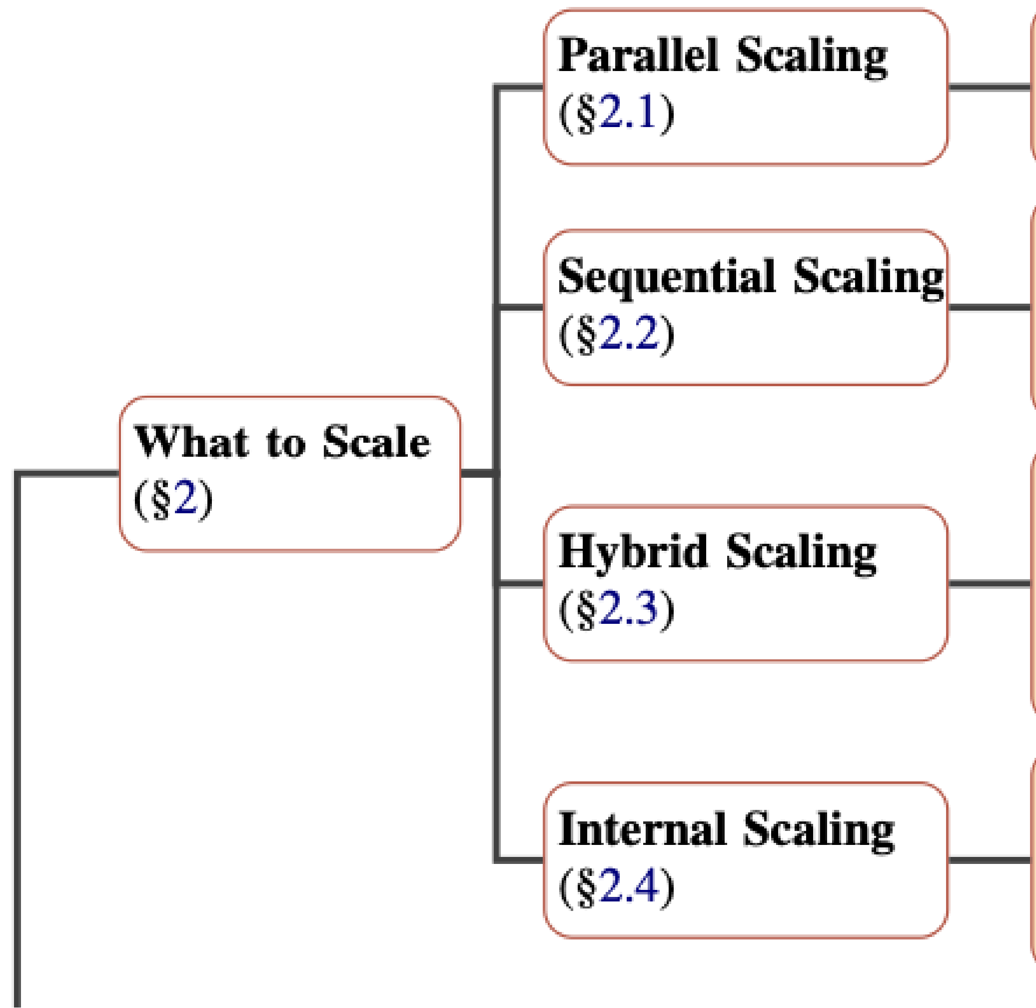
Test Time Scaling vs. Pre Training Scaling




Test-time Scaling
What to Scale (§2)
Parallel Scaling (§2.1)
Self-Consistency (Brown et al., 2024b; Irvine et al., 2023; Song et al., 2024; Snell et al., 2024; Wang et al., 2023; Nguyen et al., 2024) (Chen et al., 2024d; Wu et al., 2025b), Multi-Agents (Jiang et al., 2023), PlanSearch (Wang et al., 2024a), CCE (Zhang et al., 2025e)
Sequential Scaling (§2.2)
Self-Refine (Madaan et al., 2023; Chen et al., 2024e; Gou et al., 2024; Zhang et al., 2024d), Sequential Revision (Lee et al., 2025), ReAct (Yao et al., 2023c), Budget-aware (Kimi, 2025; Muennighoff et al., 2025; Han et al., 2025), RecurrentBlock (Geiping et al., 2025), STaR (Yuan et al., 2023; Singh et al., 2024), Meta-STaR (Xiang et al., 2025), PlanningToken (Wang et al., 2024g), RaLU (Li et al., 2025c)
Hybrid Scaling (§2.3)
MoA (Wang et al., 2025a), Tree of Thoughts (Yao et al., 2023b; Zhang et al., 2024b), Graph of Thoughts (Besta et al., 2024), Tree-Search (Chen et al., 2024g), SoS (Gandhi et al., 2024), REBASE (Wu et al., 2024c), OAIF (Guo et al., 2024), Beam-Search (Guo et al., 2024),M-CTS(Tian et al., 2024; Zhang et al., 2024e; Gao et al., 2024b; Wan et al., 2024; Chen et al., 2024a), Journey Learning(Qin et al., 2024),A-daptiveAlloc(Snell et al., 2024; Ong et al., 2025), METAL(Li et al., 2025a), rStar-Math(Guan et al., 2025a),AtomThink(Xiang et al., 2024)
Internal Scaling (§2.4)
DeepSeek-R1 (DeepSeek-AI, 2025), OpenAI-o1&o3 (OpenAI, 2024b, 2025), Gemini Flash Thinking (Google, 2024), QwQ (Qwen, 2024), K1.5 (Kimi, 2025), 3SUM (Pfau et al., 2024), OAIF (Guo et al., 2024), LIMO (Ye et al., 2025), T1 (Hou et al., 2025), Distilled-o1 (Huang et al., 2024b), RedStar (Xu et al., 2025a), SKY-T1 (NovaSky, 2025),s1 (Muennighoff et al., 2025), ITT (Hao et al., 2024)
How to Scale (§3)
Tuning (§3.1)
Supervised Finetuning (§3.1.1)
Distillation (Muennighoff et al., 2025; Huang et al., 2024b; Xu et al., 2025a; NovaSky, 2025; Bespoke, 2025) (Munkhbat et al., 2025; Ye et al., 2025), Synthesized Long CoT (Hou et al., 2025; Yeo et al., 2025), Learning Reasoning Structure (Li et al., 2025f), Long CoT warmup (Kimi, 2025) , CFT (Wang et al., 2025d)
Reinforcement Learning (§3.1.2)
Reward model-free
Rule-Based (DeepSeek-AI, 2025), cDPO (Lin et al., 2024), Focused-DPO (Zhang et al., 2025b), Selective DPO (Gao et al., 2025b), CPL (Wang et al., 2024f), OREO (Wang et al., 2024b), DAPO (Liu et al., 2024b), RFTT (Zhang et al., 2025c), SimPO (Meng et al., 2024), DQO (Ji et al., 2024), DAPO (Yu et al., 2025), VC-PPO (Yuan et al., 2025), Light-R1 (Wen et al., 2025), etc.
Reward model-based
PPO (Schulman et al., 2017), RLOO (Ahmadian et al., 2024a), GRPO (Shao et al., 2024), REINFORCE++ (Hu, 2025), DVPO (Huang et al., 2025a), PRIME (Cui et al., 2025), SPPD (Yi et al., 2025), etc.
Inference (§3.2)
Stimulation (§3.2.1)
Prompt Strategy
Hint-infer (Li et al., 2025b), Dipper (Lau et al., 2024), EVA (Ye et al., 2024), EvalPlan(Saha et al., 2025), ReasonFlux (Yang et al., 2025a), Hong et al. (2024)
Decode Strategy
Filler Tokens (Pfau et al., 2024), Budget Forcing (Muennighoff et al., 2025), LTV (Kong et al., 2025), AFT (Li et al., 2025g), Predictive-Decoding (Ma et al., 2025a)
Self-Repetition
Self-Consistency (Wang et al., 2023), Self-Refine (Madaan et al., 2023), DeCRIM (Ferraz et al., 2024), CCE (Zhang et al., 2025e), TreeBoN (Qiu et al., 2024)
Mixture-of-Model
MoA (Wang et al., 2025a), RR-MP (He et al., 2025), BRAIN (Chen et al., 2024f)
Verification (§3.2.2)
Outcome
Output Verification (Cobbe et al., 2021), Generative Verifier (Zhang et al., 2025d), Self-Reflection Feedback (Li et al., 2025h), Discriminator (Chen et al., 2024g), OVM (Yu et al., 2024b), Heuristic (DeepSeek-AI, 2025), Bandit (Sui et al., 2025), Functional (Lee et al., 2025), XoT (Liu et al., 2023b), WoT (Zhang et al., 2024c)
Process
State Evaluator (Yao et al., 2023b; Zhang et al., 2024b), SIaM (Yu et al., 2024a), Deductive Verification (Ling et al., 2023), Self-Evaluator (Xie et al., 2023), V-STaR (Hosseini et al., 2024), Tool (Li et al., 2025b), PoT (Chen et al., 2023a)
Search (§3.2.3)
TreeSearch (Yao et al., 2023b; Chen et al., 2024g),GraphSearch (Besta et al., 2024),C-MSTS (Lin et al., 2025), MCTS (Tian et al., 2024; Zhang et al., 2024e; Gao et al., 2024b; Wan et al., 2024; Chen et al., 2024a), SPaR (Cheng et al., 2025), REBASE (Wu et al., 2024c), SoS (Gandhi et al., 2024), CoAT (Pan et al., 2025a),Beam-Search (Guo et al., 2024; Xie et al., 2023), Lookahead-Search (Snell et al., 2024; Zhang et al., 2023b), etc.
Aggregation (§3.2.4)
Selection
Majority Voting(Wang et al., 2023; Chen et al., 2024d), BOND(Sessa et al., 2024), Filter Vote(Chen et al., 2024d), Length-filtered Vote(Wu et al., 2025b), Best-of-N (Irvine et al., 2023; Song et al., 2024), Rejection Sampling (Kimi, 2025), etc.
Fusion
BoN (weighted) (Brown et al., 2024b), Synthesize (Wang et al., 2025a), etc.

What & How: Summary




What to Scale
(§2)
Parallel Scaling
(§2.1)
Sequential Scaling
(§2.2)
Hybrid Scaling
(§2.3)
Internal Scaling
(§2.4)

What to Scale: Parallel Scaling

- What is being scaled at the inference stage?
- Parallel Scaling
  - Generating multiple outputs in parallel and then aggregating them into a final answer.
  - Can be from multiple models, or the same model run repeatedly
  - Same model adjustment from hyperparameter adjustment or prompt rephrasing

- Effectiveness derives from:
  - Coverage: the likelihood of generating at least one correct response
  - Aggregation Quality: if a correct response is successfully identified
  - Idea that complex solutions have multiple pathways to the answer

What to Scale: Sequential & Hybrid

- Sequential Scaling
  - Involves explicitly directing later computations based on intermediate steps.
  - Has several states that involve previous states and problem context
  - Chain-of-Thought (CoT) Prompting, Step-by-Step, Refine
  - Iterations create self-correction, improving accuracy
- Hybrid Scaling
  - Combines Sequential and Parallel Scaling
  - Generate multiple hypotheses and then refine/evaluate them
  - Early work: Tree of Thoughts, Graph of Thoughts
  - More advanced: Monte Carlo Tree Search, Multi-Agent Reasoning (debate)

e.g. Sequential scaling Performance

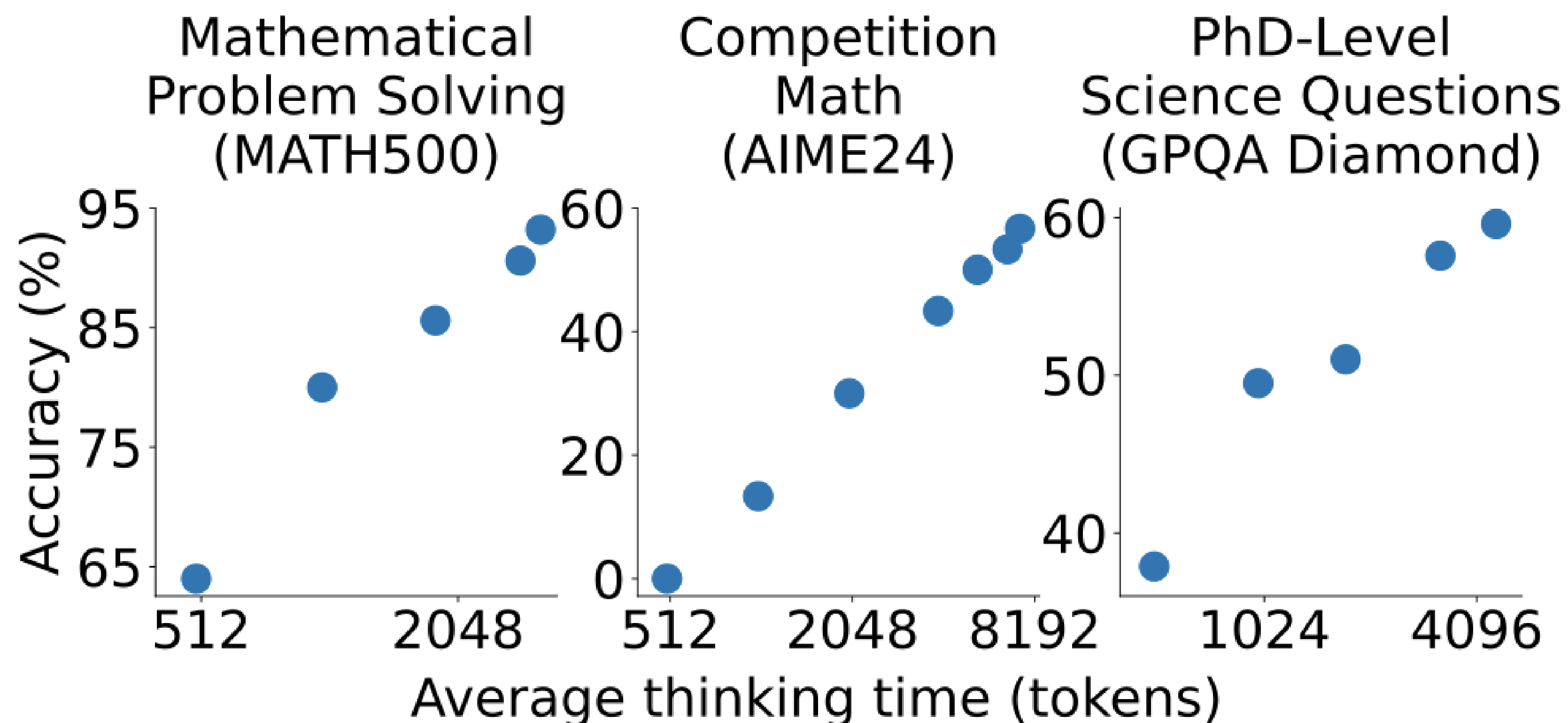


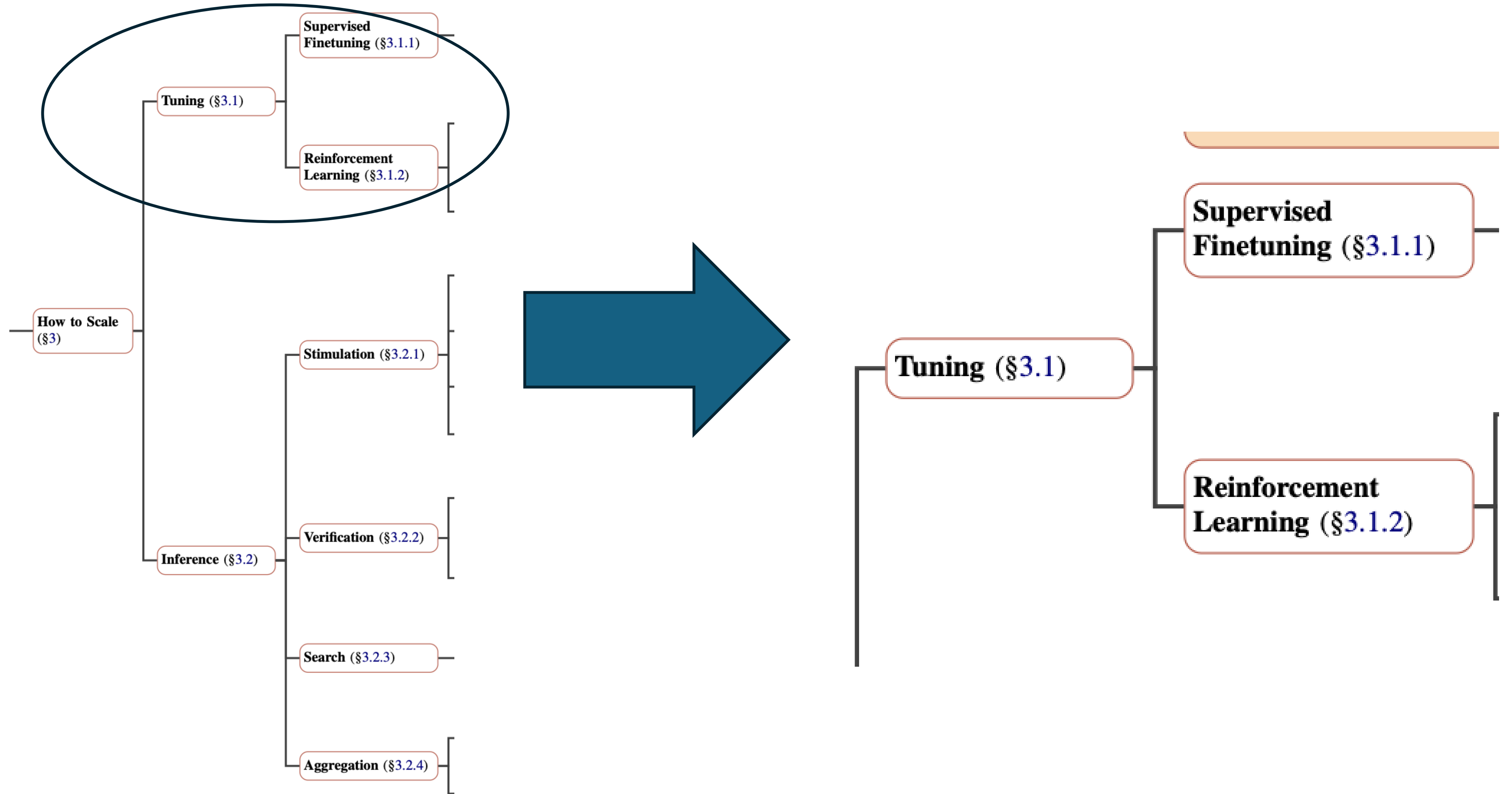
Figure 1. **Test-time scaling with s1-32B.** We benchmark **s1-32B** on reasoning-intensive tasks and vary test-time compute.

What to Scale: Internal Scaling (->how much?)

- Internal Scaling
  - Model chooses how much scaling to do for the problem instead of human strategy
- Param Trainable Model
  - Continuously update the model based on reasoning tasks via some training procedure
    - long CoT examples produced by external scaling
  - Outcome-oriented reward modeling for RL (DeepSeek)
- Frozen Model
  - At Test Time, the model generates a sequence of internal states (z)

$$z_{t+1} = f_\theta(z_t), \quad \mathrm{stop}(z_t) = \pi_\theta(z_t).$$

  - Controls when to stop via learned policy
  - Leads to emergent thinking without external prompting


Supervised
Finetuning (§3.1.1)
Tuning (§3.1)
Reinforcement
Learning (§3.1.2)
How to Scale
(§3)
Stimulation (§3.2.1)
Verification (§3.2.2)
Inference (§3.2)
Search (§3.2.3)
Aggregation (§3.2.4)
Supervised
Finetuning (§3.1.1)
Tuning (§3.1)
Reinforcement
Learning (§3.1.2)

How to Scale: Tuning Based Approaches

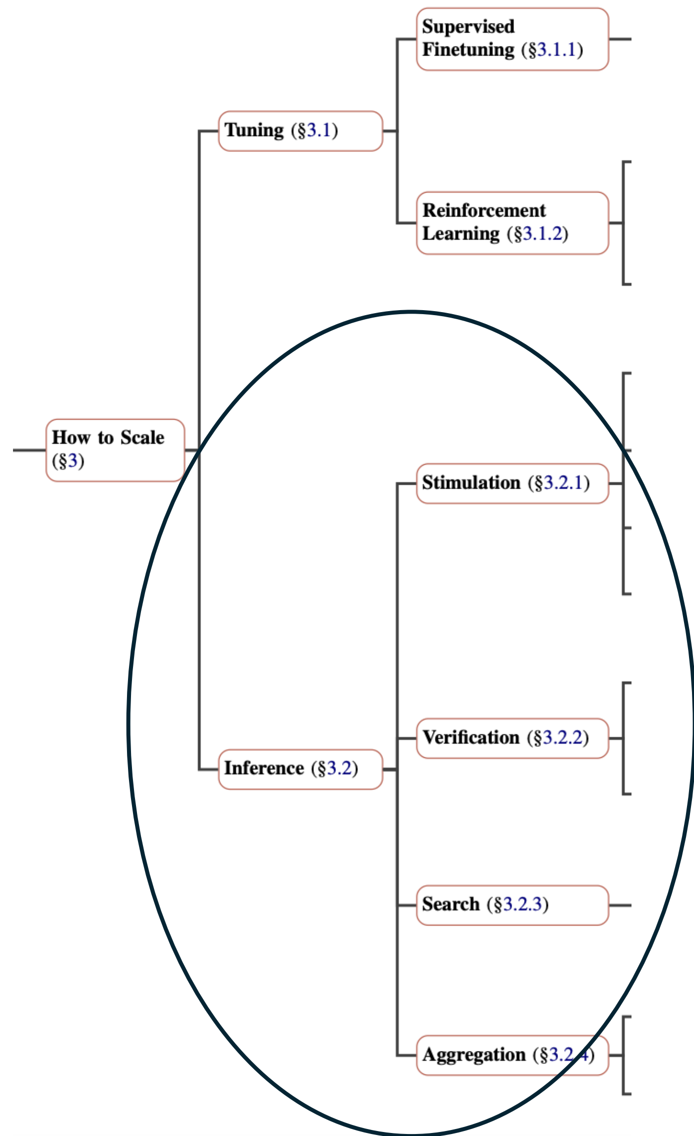
- Directly tuning the LLM's parameters with 2 approaches: SFT and RL
- 1. Supervised Fine Tuning (SFT)
  - Train LLM to mimic the rationale/structure to prompt the model to think through complex problems
  - 2 main approaches
- SFT Imitation
  - Generate long CoT demonstrations using test-time “planner” algorithms and then fine-tune the model to imitate those demonstrations
  - STaR: Can be guided by the model itself (generates step-by-step solutions with filtering/verification)
  - ReST-MCTS: use MCTS planner to model itself to reasoning steps
- SFT Distillation
  - Use responses of "stronger" models for supervised learning
  - Can lead to smaller models answer questions just as well as the teacher model

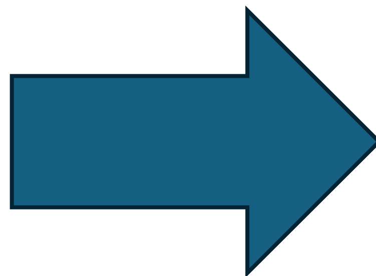
How to Scale: Tuning Based Approaches

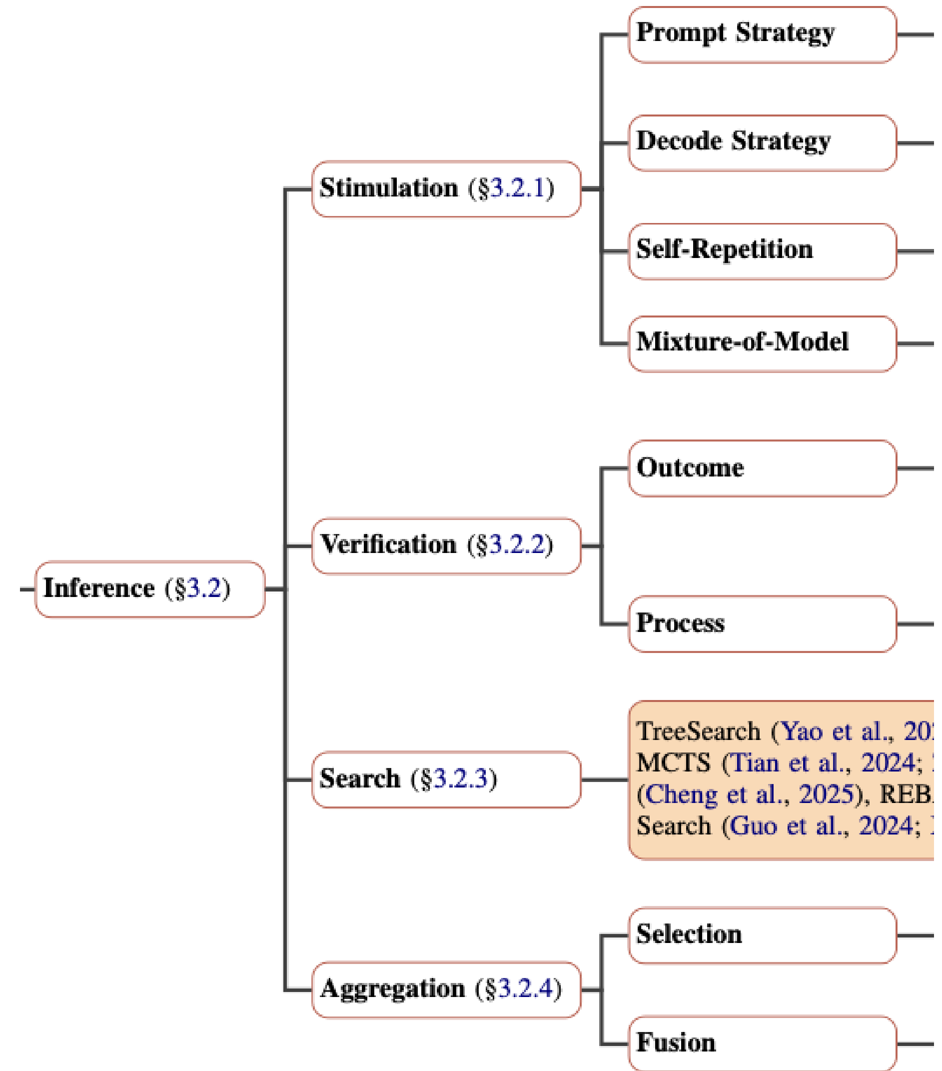
- 2. RL: Reward Model-Free
  - Verifiable reward by DeepSeek R1: rule-based reward mechanisms to optimize accuracy in large models
    - SimpleR1: Open-source reproduction of R1
  - OpenR1: HuggingFace's open-source tool for RL
  - cDPO: preference-based optimizer, utilizing critical tokens (base for many other expansions)
  - OREO: value-based optimizer for mathematical reasoning

- 2. RL: Reward Model-Based
  - PPO: Using Human Based model for optimization
    - ReMax takes PPO and reduces hyperparameters, compute time, and need for additional value models
      - Reinforce/Reinforce++ also do this, ReMax more greedy
  - UGDA: refines reward model with (previously) uncertain data.

https://arxiv.org/pdf/2411.19943


Supervised Finetuning (§3.1.1)
Tuning (§3.1)
Reinforcement Learning (§3.1.2)
How to Scale (§3)
Stimulation (§3.2.1)
Verification (§3.2.2)
Inference (§3.2)
Search (§3.2.3)
Aggregation (§3.2.4)




Prompt Strategy
Decode Strategy
Stimulation (§3.2.1)
Self-Repetition
Mixture-of-Model
Outcome
Verification (§3.2.2)
Inference (§3.2)
Process
Search (§3.2.3)
Selection
Aggregation (§3.2.4)
Fusion

How to Scale: Inference Based Approach

- Dynamically adjust parameters during deployment/inference
- Stimulation
  - Getting LLM to think more and allocate longer samples
  - Prompting Strategies
    - "Think Step by Step," and listing requirements to stimulate more samples
  - Decoding Strategies
    - Input more filler phrases or tokens, enforcing intermediate generation (drafts), enforcing prior distributions of latent vectors
  - Self-Repetition Strategies
    - Prompt LLM repeatedly during decoding stage, another is to mimic refinement process
  - Mixture of Model Strategies
    - Ask different models about what they think. Can be all the same or different perspectives

How to Scale: Inference Based Approach

- Verification
  - How do we make sure the LLM is generating a correct response?
    - Can be used for Parallel Scaling, Sequential (to know when to stop), Aggregation or Searching Process (we'll get back to this)
  - Outcome Verification
    - Model Voting
    - Self-consistency
    - Separate algorithms/functions (verifiers)
    - Code generation checks
    - Separate LLM verifier (Judges), Agents
    - RAG
  - Process Verification
    - AKA: Process Reward Model, State Verification
    - Evaluating if the process is correct: Is it actually using CoT? Do the steps to reach the outcome make sense?
    - Process Verification harder for LLMs to evaluate if too complex or long context, decomposition needed
    - Used mostly in Code Generation or Mathmatical Reasoning

How to Scale: Inference Based Approach

- Search
  - Make sure LLM is utilizing its vast database of knowledge to ensure accuracy
  - Can organize thoughts into a tree and utilize BFS or DFS
  - Utilize Monte Carlo Tree Search during decoding to guide planning
  - Graph Search is also experimented, utilizing stochastic beam search
- Aggregation
  - How to consolidate multiple answers
  - Selection
    - Self-consistency of different routes (most common answer, but sometimes filtering required), Selection Agent
    - Best-of-N (score based on external verifier)
  - Fusion
    - Combine Best-of-N (based on external verifier)
    - Have LLM summarize

Scaling via Verifiers: Search Methods

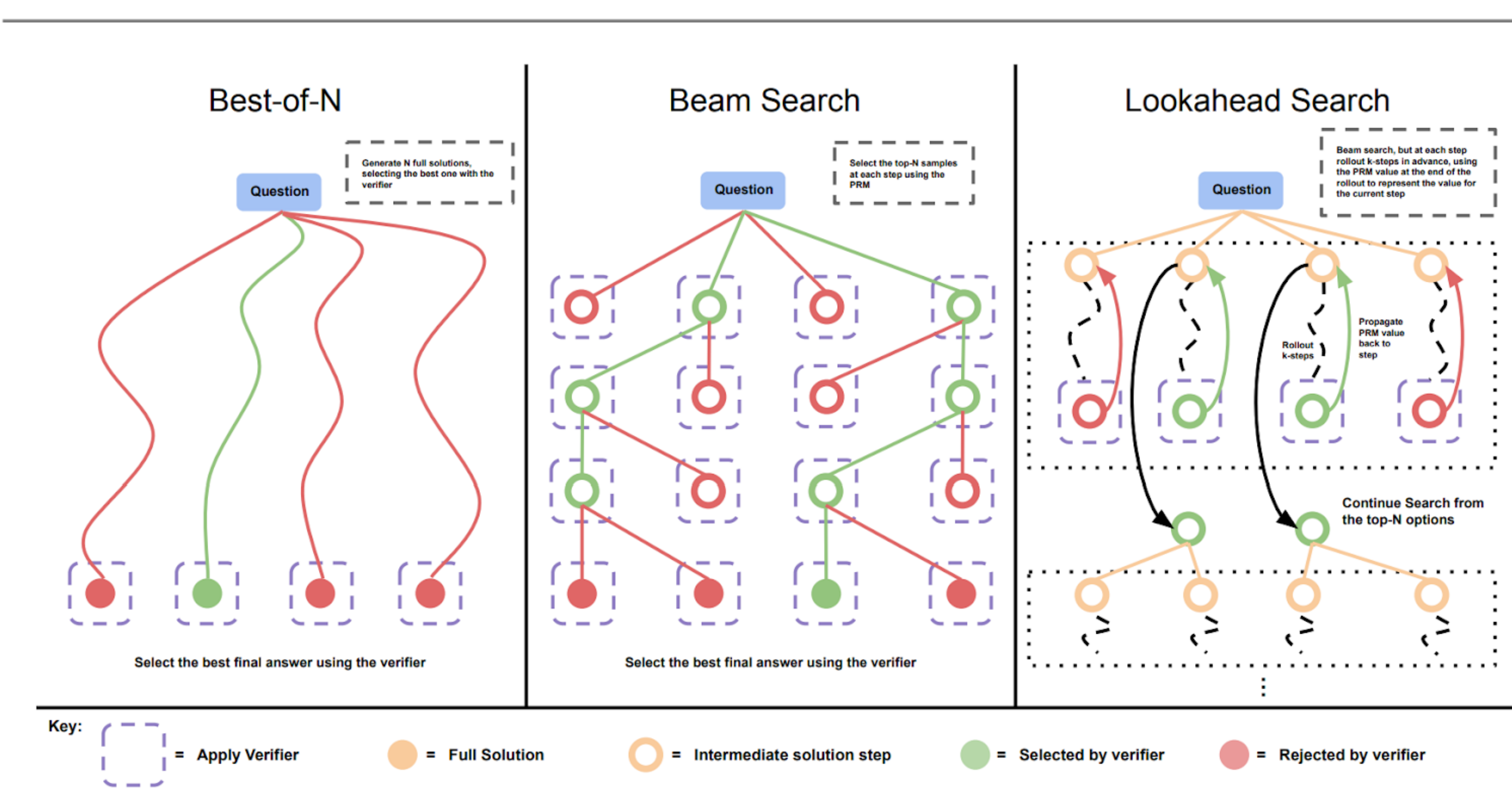
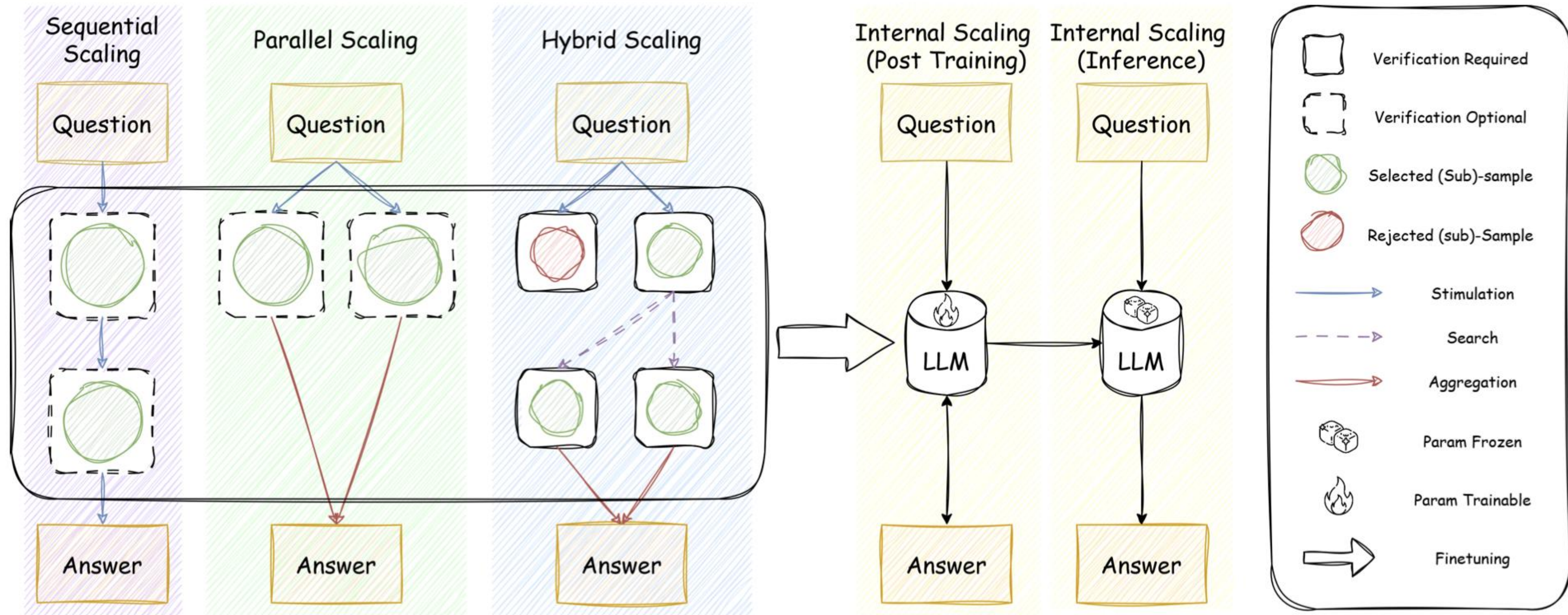


Figure 2 | ***Comparing different PRM search methods.*** **Left:** Best-of-N samples N full answers and then selects the best answer according to the PRM final score. **Center:** Beam search samples N candidates at each step, and selects the top M according to the PRM to continue the search from. **Right:** lookahead-search extends each step in beam-search to utilize a k-step lookahead while assessing which steps to retain and continue the search from. Thus lookahead-search needs more compute.

e.g., Tree-Based Test Time Inference Search

- Why sampling isn't enough:
  - Sampling generates full completions blindly
  - It lacks structure and wastes compute on bad outputs
- Tree search:
  - Builds solutions incrementally (step-by-step)
  - Allows dynamic allocation of compute to promising paths
- Example: MCTS (Monte Carlo Tree Search)
  - Simulates many possible completions (rollouts)
  - Assigns rewards based on full solution outcomes
  - Backpropagates rewards to improve search decisions
- Drawback:
  - MCTS is compute-heavy – expensive rollouts for every path
  - Doesn’t scale well for LLM inference, especially with long solutions

What & How: Summary



Organization and Trends

- 2022 – 2023
  - Emphasized structured inference to guide LLMs
- 2024
  - Methods like PRM and MCTS enabled automatic supervision of reasoning
- 2025
  - Pure RL can also elicit comprehensive, sound reasoning

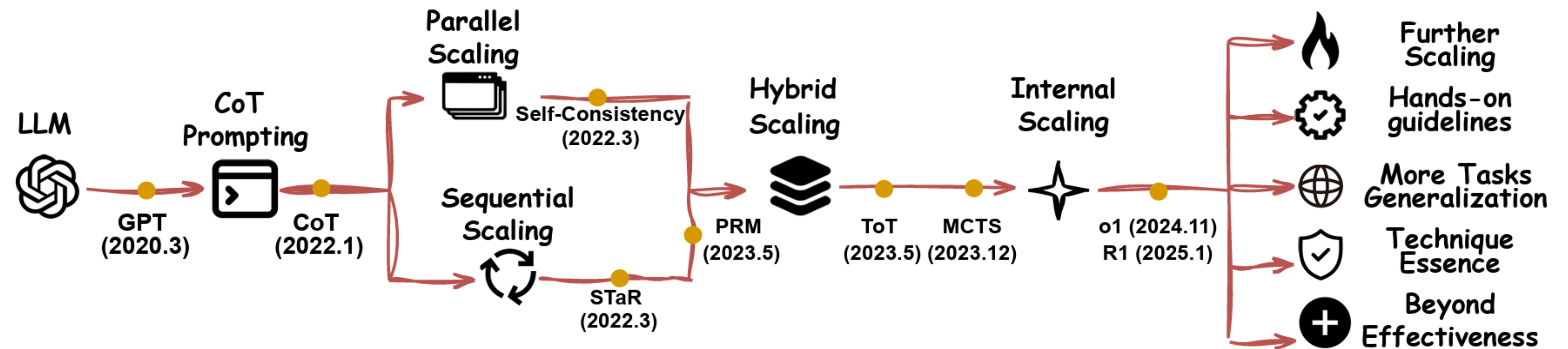


Figure 4: From Emergence to the Next Frontier, the Evolutionary Path of Test-Time Scaling.

https://openreview.net/forum?id=VTF8yNQM66

Three papers from the student team

- **A programming strategy to enable easier Search for TTS**

- **Agentic Planning**
  - **RAP** { MCTS, State / world model, Search, Action Planning }
  - **RL-R1:** Policy RL to train the ability for testing planning
- **Agentic Reasoning**
  - **ReAct:** [observe] → Act → Reason
  - **Model+ Reasoning:** World Model → Reason & Plan

Enhancing Agent Programming with Search Over Program Execution Paths

Authors: Zhening Li, Armando Solar-Lezama, Yisong Yue, Stephan Zheng

NeurIPS 2025

Overview

EnCompass is a programming framework for adding **inference-time strategies** to **AI agents**.

sampling, refinement, backtracking, tree search, etc. · *any program that calls LLMs to solve subtasks*

EnCompass enables experimenting with different inference-time strategies **without modifying** the underlying agent source code.

Here's a program that makes LLM calls to solve tasks in an escape room:

```
def solve_escape_room():
    cipher = solve_cipher()
    logic = solve_logic(cipher)
    riddle = solve_riddle(cipher, logic)
    code = solve_combination(cipher,
logic, riddle)
    open_door(code)
```

But LLMs make mistakes — let's use inference-time strategies like resampling and backtracking.

```
def solve_escape_room():
    # Try the cipher multiple times
    cipher_solutions = []
    for attempt in range(N):
        candidate = llm.solve_cipher()
        score = verify_cipher(candidate)
        cipher_solutions.append((candidate, score))

    best_cipher, best_cipher_score = max(...)

    # Now try the logic puzzle multiple times
    ...
    best_logic, best_logic_score = max(...)
    if best_logic_score == 0:
        # Backtrack to attempt cipher again
        candidate = llm.solve_cipher()
```

The inference-time strategy is hardcoded into the workflow.

❌ readable
❌ modular
❌ flexible
❌ scalable

The dream:

1. Annotate the steps that we may resample or backtrack to

2. Annotate information used by the resampling/backtracking strategy

3. Resampling/backtracking happens automatically at runtime!

The inference-time strategy is separated from the workflow.

- ✅ readable
- ✅ modular
- ✅ flexible
- ✅ scalable

EnCompass: Separate inference-time strategies from the workflow

```python
def solve_escape_room():
    cipher = solve_cipher()
    logic = solve_logic(cipher)
    riddle = solve_riddle(cipher, logic)
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)

solve_escape_room()
```

EnCompass: Separate inference-time strategies from the workflow

```python
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)

solve_escape_room().search(search_algo, **search_config)
```

EnCompass: Separate inference-time strategies from the workflow

```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)

solve_escape_room().search(search_algo, **search_config)
```

Inference time strategies as search over program execution paths

EnCompass: Separate inference-time strategies from the workflow

```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))  ✅ maximize recorded score
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))  ✅ maximize recorded score
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))  ✅ maximize recorded score
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)  ✅ maximize recorded score

solve_escape_room().search(search_algo, **search_config)
```

EnCompass: Separate inference-time strategies from the workflow

```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))  ✅ maximize recorded score
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))  ✅ maximize recorded score
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))  ✅ maximize recorded score
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)  ✅ maximize recorded score

solve_escape_room().search(search_algo, **search_config)
```

EnCompass: Separate inference-time strategies from the workflow

```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)

solve_escape_room().search(search_algo, **search_config)
```

✅

EnCompass: Separate inference-time strategies from the workflow

Global best-of-N

```python
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)

solve_escape_room().search("sampling", num_rollouts=5)
```

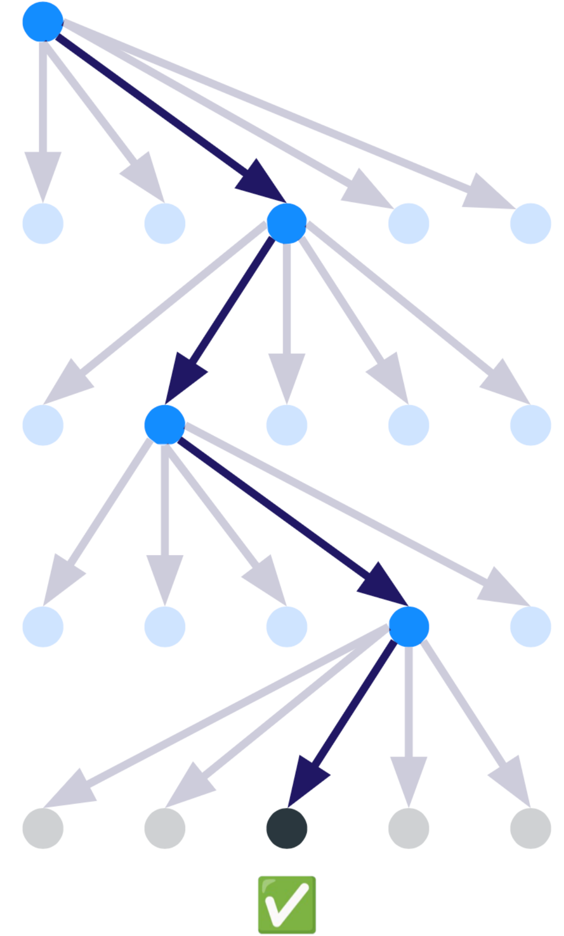
✅

EnCompass: Separate inference-time strategies from the workflow

```python
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle))
    branchpoint()
    code = solve_combination(cipher, logic, riddle)
    success = open_door(code)
    record_score(success)

solve_escape_room().search("beam", beam_width=1, branching=5)
```

Local best-of-N



Inference time strategies as search over program execution paths

```
@encompass.compile
def solve_escape_room():
    branchpoint()

solve_escape_room().search("beam", beam_width=1,
branching=5)
```

Local best-of-N

Inference time strategies as search over program execution paths

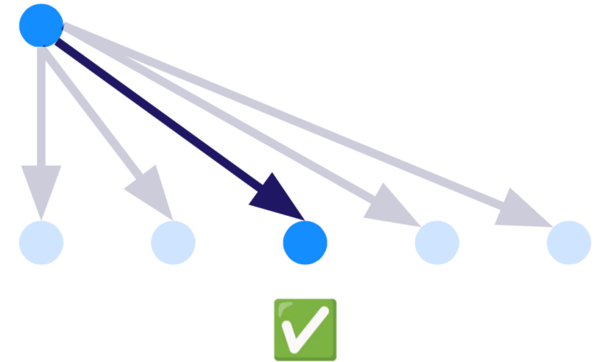
```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
```

✅ maximize recorded score

```
    branchpoint()

solve_escape_room().search("beam", beam_width=1, branching=5)
```

Local best-of-N

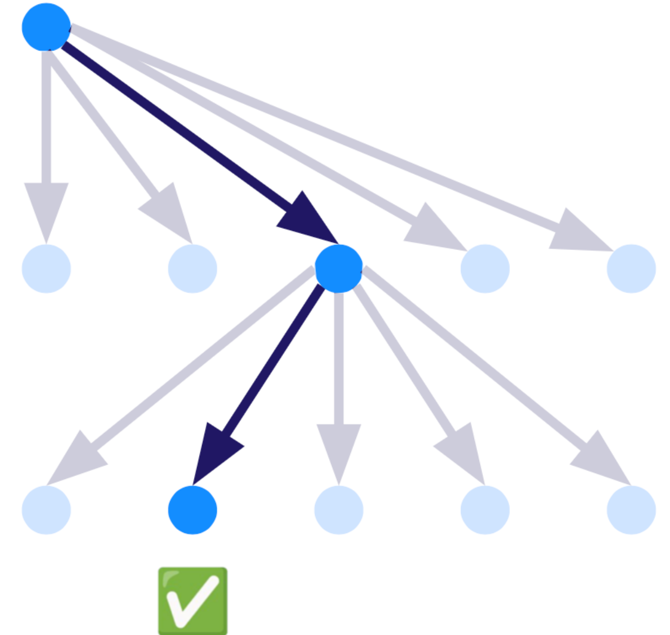


Inference time strategies as search over program execution paths

```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic)) ✅ maximize recorded score
    branchpoint()

solve_escape_room().search("beam", beam_width=1, branching=5)
```

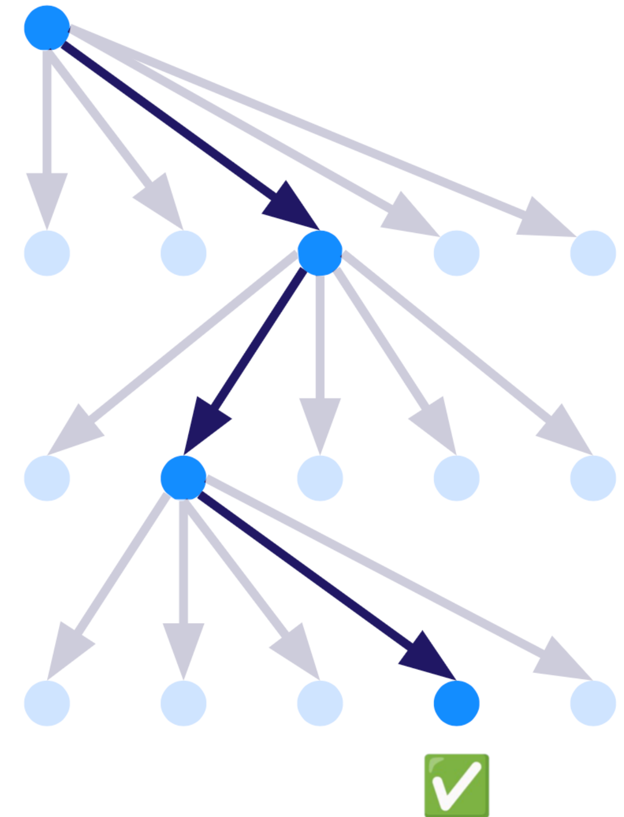
Local best-of-N



Inference time strategies as search over program execution paths

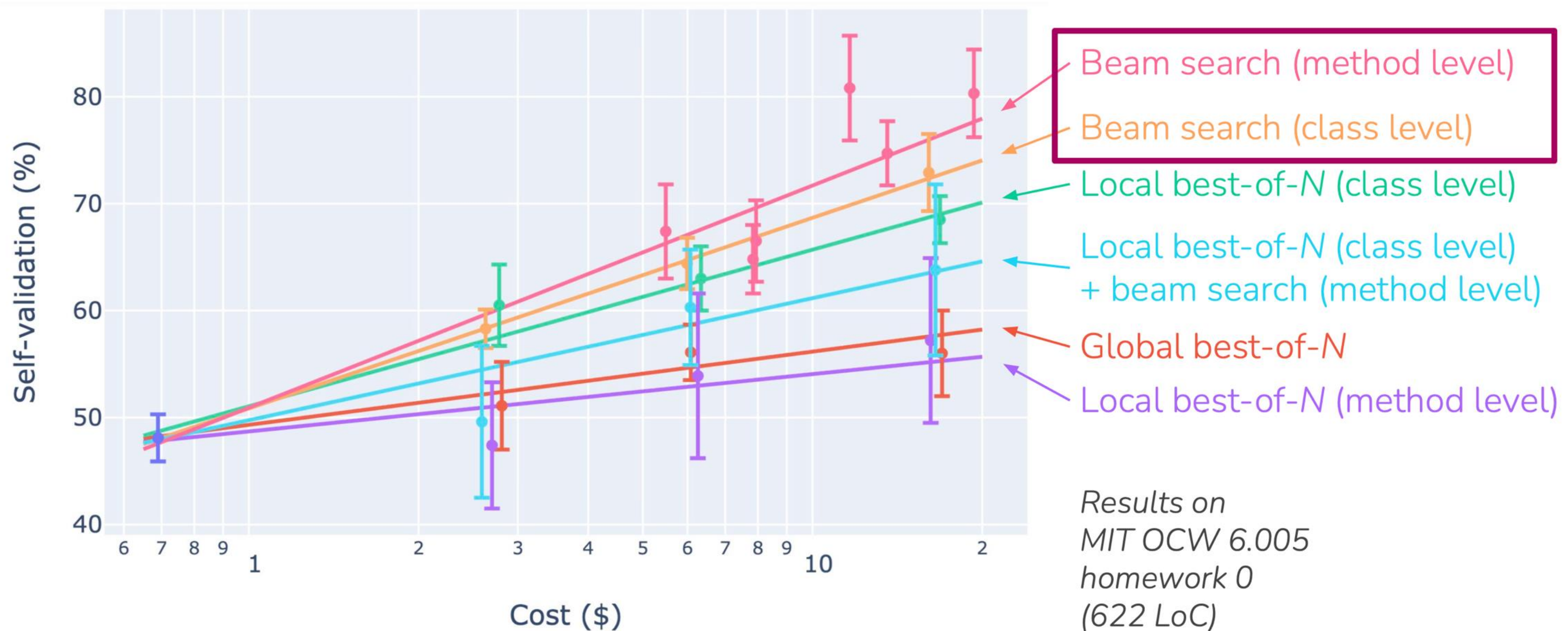
```
@encompass.compile
def solve_escape_room():
    branchpoint()
    cipher = solve_cipher()
    record_score(verify_cipher(cipher))
    branchpoint()
    logic = solve_logic(cipher)
    record_score(verify_logic(logic))
    branchpoint()
    riddle = solve_riddle(cipher, logic)
    record_score(verify_riddle(riddle)) ✅ maximize recorded
score
    branchpoint()

solve_escape_room().search("beam", beam_width=1,
```



Case Study: Java → Python repository translation agent

Agent with 597 lines of code (not including helper/utility functions, etc.)
•Iterates through each class and method in Java repo, translating methods one by



Zhening Li, Armando Solar-Lezama, Yisong Yue and Stephan Zheng. “EnCompass: Enhancing Agent Programming with Search Over Program Execution Paths.” NeurIPS, 2025.

Case Study: Java → Python repository translation agent

Base Agent (5 LLM calls per method):

- Write Python skeleton → Translate methods → Generate tests → Run Java → Compare outputs

ENCOMPASS Modification:

- Add **branchpoint()** before each LLM call → Experiment with 6 strategy combos

Key Result:

- **Beam(coarse) + Beam(fine)** scales best ($p < 0.03$ vs all others)
- Performance scales **log-linearly** with cost (all χ^2 p-values > 0.3)

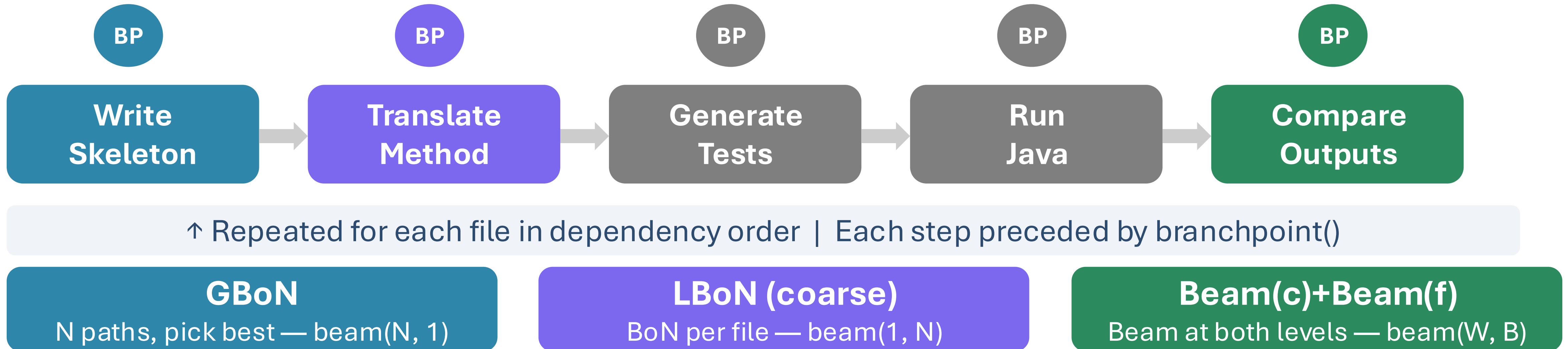
Code Savings:

- **ENCOMPASS: +75 lines, 1 new function**
- **Plain Python: +423 lines, 20 functions, 189 indent changes**

Key takeaways

- EnCompass separates the overlaying inference-time strategy from the underlying workflow
- This separation of concerns enables independent experimentation of each component
- This facilitates the discovery of inference-time strategies that scale better

CS1 Workflow: File-by-File Translation with 5 Branchpoints



Result: Beam(coarse) + Beam(fine) scales best ($p < 0.03$). Log-linear scaling with cost. Applied to 4 additional repos (5,756 LoC) — consistently outperforms simpler strategies.

CS1 Code: Repo Translation — ENCOMPASS vs Plain Python

ENCOMPASS — +12 lines

```
@encompass.compile
def translate_functions(source):
  for source_fn in source:
    branchpoint()
    target_fn = translate(source_fn)
    compile_success = compile_(target_fn)
    record_score(compile_success)

    branchpoint()
    test_score = run_unit_test(target_fn)
    record_score(test_score)
```

Plain Python — +423 lines, 20 functions

```
class State(Enum):
  TRANSLATE = auto()
  UNIT_TEST = auto()

def step(state, frame):
  frame = frame.copy()
  if state == State.TRANSLATE:
    frame["target"] = translate(
        frame["source_fn"])
    score = compile_(frame["target"])
    return State.UNIT_TEST, frame, score
  # ... 189 indent changes, 20 new fns
```

Key: ENCOMPASS preserves the for-loop structure. Plain Python unrolls it into a state machine with frame dictionaries — losing readability, type safety, and linter support.

CS 1: Cross-Repository Generalization

Validated on ps0 (622 LoC), then tested on ps1–ps4 (5,756 LoC total)

- Each repository: 1,100–1,900 lines of Java, cost $13–$39 per run

Results (self-validation % — cost controlled):

- **Beam(c) + Beam(f)** consistently outperforms simpler strategies across all repos
- Beam width 2 (file) + beam width 3 (method) vs N=16 for GBoN/LBoN

Key Takeaway:

- The **best-performing strategy is also the hardest to implement** in plain Python
- ENCOMPASS makes it trivial: just change search parameters

CS 2: Hypothesis Search for ARC-AGI

Simple 2-step agent with 2 branchpoints:

- Step 1: Generate hypothesis (transformation rule)
- Step 2: Generate Python code implementing the rule

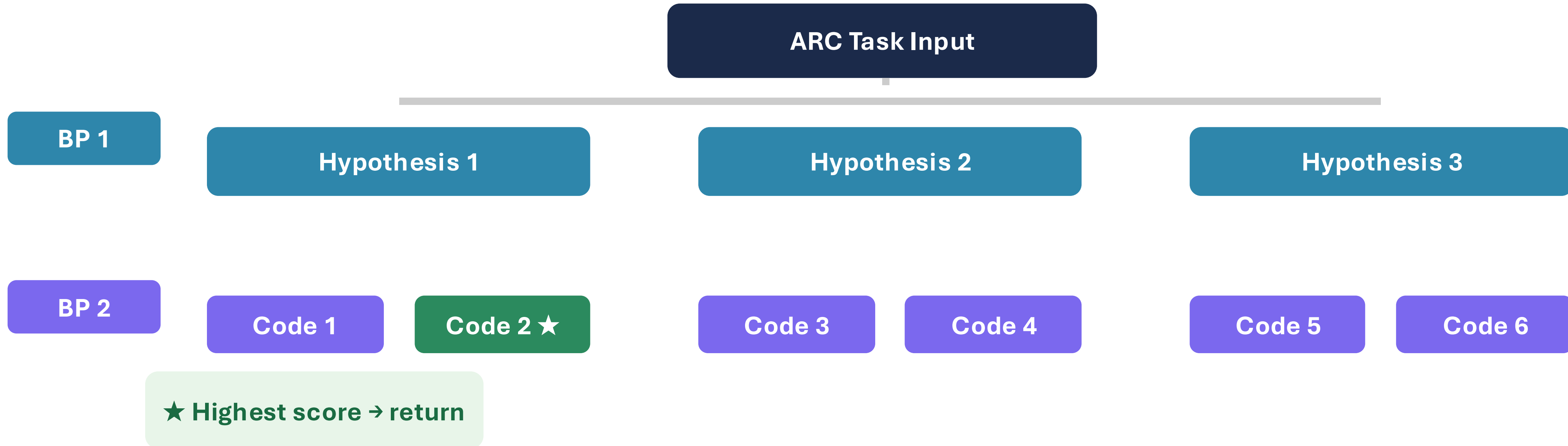
Strategies compared:

- BFS (multithreaded, out-of-the-box) vs. Global Best-of-N
- Result: BFS ≈ GBoN on ARC subset (reproduces parallelized Hypothesis Search)

Code Savings:

- **ENCOMPASS: +8 lines, 0 new functions, 0 indent changes**
- **Plain Python: +21 lines, 2 new functions, 10 indent changes**
- Even a simple agent becomes noticeably harder to read without ENCOMPASS

CS2 Workflow: Two-Step Search Tree for ARC-AGI



ARC-AGI Results (GPT-4o, 60 tasks):
Base: 24.0% → GBoN(N=8): 36.3% → BFS(branching=8): 38.3% | BFS ≈ GBoN — both effective, trivial to switch
Outperforms ADAS best agent (32.7%) discovered through costly meta-agent search

CS2 Code: ARC-AGI Hypothesis Search

Base Agent (Listing 1) — 9 lines

```
def two_step_agent(task_info):
  # Step 1: Get NL hypothesis
  hypothesis = hypothesis_agent(
      [task_info], instruction)
  # Step 2: Implement in code
  code = solver_agent(
      [task_info, hypothesis],
      solver_instruction)
  return get_test_output(code)
```

ENCOMPASS (Listing 2) — +8 lines

```
@encompass.compile
def two_step_agent(task_info):
  branchpoint()
  hypothesis = hypothesis_agent(
      [task_info], instruction)
  branchpoint()
  code = solver_agent(
      [task_info, hypothesis], ...)
  percent, fb = run_validation(code)
  record_score(n_correct)
  if percent == 1.0:
    early_stop_search()
  return get_test_output(code)
```

Plain Python (Listing 3) — +21 lines

```
from concurrent.futures import
    ThreadPoolExecutor, as_completed

def two_step_agent(task_info, br):
  results = []
  full_solved = False
  with ThreadPoolExecutor() as ex:
    def run_one_forward_pass():
      if full_solved: return
      hypothesis =
hypothesis_agent(...)
      def implement_in_code():
        nonlocal full_solved
        code = solver_agent(...)
        # ... nested futures
  # Workflow obscured by concurrency
```

Progression: Base agent is clean and simple. ENCOMPASS adds 8 lines preserving structure. Plain Python requires ThreadPoolExecutor, nested closures, and nonlocal — obscuring the 2-step workflow.

CS 3: Reflexion on LeetCodeHard

Base Agent: Refinement loop (generate → test → reflect → retry)

- 2 branchpoints enable GBoN and re-expand best-first search

Key Finding — External Search > Refinement Loops:

- Reflexion k=5 iterations: ~0.40 pass@1
- **Best-first search (10 steps): ~0.46 pass@1**
- Increasing N or search steps scales more cost-efficiently than more iterations

Code Savings:

- **ENCOMPASS: +9 lines, 0 new functions**
- **Plain Python: +27 lines, 2 new functions, 8 indent changes**

Takeaway: A simple strategy — repeatedly expanding the best state — can work remarkably well

CS3 Workflow: Refinement vs External Search Scaling

Vanilla Reflexion: Refinement Loop

Generate → Test → Reflect → Retry

↻ repeat k times

k=1: 0.34 | k=3: 0.39 | k=5: 0.40 pass@1
Diminishing returns: +5 iters → only +0.06

ENCOMPASS: External Search

BP 1: Generate

BP 2: Reflect | BP 2: Reflect | BP 2: Reflect

Best-first: always expand highest-scoring state

GBoN(5): 0.43 | BeFS(5): 0.44 | BeFS(10): 0.46
Stronger scaling: search steps → more efficient than loop iterations

Key Insight: Refinement hits diminishing returns (0.34 → 0.40 over 5x more iterations). External search via ENCOMPASS reaches 0.46 — scaling more cost-efficiently. A simple strategy (always expand best state) works remarkably well.

CS3 Code: Reflexion Agent — Base vs ENCOMPASS

Base Reflexion Agent (Listing 24) — 23 lines

```
def reflexion_agent(task, tests, max_iters):
  code = solver_agent(task)
  pct, fb = run_validation(code, tests)
  if pct == 1.0: return code

  for cur_iter in range(1, max_iters):
    reflection = self_reflection_agent(
        code, fb)
    code = solver_agent(task, code,
        fb, reflection)
    pct, fb = run_validation(code, tests)
    if pct == 1.0: return code
  return code
```

ENCOMPASS (Listing 25) — +9 lines

```
@encompass.compile
def reflexion_agent(task, tests, max_iters):
  record_score(0.2)
  branchpoint()              # BP 1
  code = solver_agent(task)
  pct, fb = run_validation(code, tests)
  record_score(pct)
  optional_return(code)
  if pct == 1.0:
    early_stop_search()

  for cur_iter in range(1, max_iters):
    branchpoint()            # BP 2
    reflection = self_reflection_agent(
        code, fb)
    code = solver_agent(task, code, ...)
    pct, fb = run_validation(code, tests)
    record_score(pct)
    optional_return(code)
    if pct == 1.0:
      early_stop_search()
  return code
```

Teal lines = additions to base code. Structure unchanged. Search invoked via:
`reflexion_agent(...).search("reexpand_best_first", max_num_results=5)`

Code Savings: 3–6x Reduction (Table 1)

Case Study	Approach	Lines Added	New Functions	Indent Changes
1. Code Repo Translation	-ENCOMPASS	**+423**	**+20**	**189**
(LoC = 597)	+ENCOMPASS	**+75**	**+1**	**0**
2. Hypothesis Search	-ENCOMPASS	**+21**	**+2**	**10**
(LoC = 11)	+ENCOMPASS	**+8**	**+0**	**0**
3. Reflexion	-ENCOMPASS	**+27**	**+2**	**8**
(LoC = 20)	+ENCOMPASS	**+9**	**+0**	**0**