PaT: Planning-after-Trial for Efficient Test-Time Code Generation

Youngsik Yoon[1], **Sungjae Lee**[1], **Seockbean Song**[2], **Siwei Wang**[3], **Wei Chen**[3], **Jungseul Ok**[1,2]*
[1]Department of Computer Science and Engineering, POSTECH, South Korea
[2]Graduate School of Artificial Intelligence, POSTECH, South Korea
[3]Microsoft Research Asia, Beijing, China
{ysyoon97, sungjaelee25, shinebobo, jungseul.ok}@postech.ac.kr,
{siweiwang, weic}@microsoft.com

Abstract

Beyond training-time optimization, scaling test-time computation has emerged as a key paradigm to extend the reasoning capabilities of Large Language Models (LLMs). However, most existing methods adopt a rigid Planning-before-Trial (PbT) policy, which inefficiently allocates test-time compute by incurring planning overhead even on directly solvable problems. We propose Planning-after-Trial (PaT), an adaptive policy for code generation that invokes a planner only upon verification failure. This adaptive policy naturally enables a heterogeneous model configuration: a cost-efficient model handles generation attempts, while a powerful model is reserved for targeted planning interventions. Empirically, across multiple benchmarks and model families, our approach significantly advances the cost-performance Pareto frontier. Notably, our heterogeneous configuration achieves performance comparable to a large homogeneous model while reducing inference cost by approximately 69%.

1 Introduction

Large Language Models (LLMs) have achieved remarkable success in code generation, driven by massive model and data scaling (Chen et al., 2021; Li et al., 2022; OpenAI, 2023). Beyond scaling, conventional approaches focused on training-time strategies, such as Supervised Fine-Tuning (SFT) (Roziere et al., 2023; Luo et al., 2023) and Reinforcement Learning from Human Feedback (RLHF) (Ouyang et al., 2022). Complementing these efforts, recent research scales test-time computation to handle complex algorithmic logic (Wei et al., 2022; Yao et al., 2023; Snell et al., 2025). In particular, advancements have been made to simulate human-like coding behavior by decomposing complex problems into manageable sub-problems (Chen et al., 2024a; Le et al., 2024) rather than generating the solution directly.

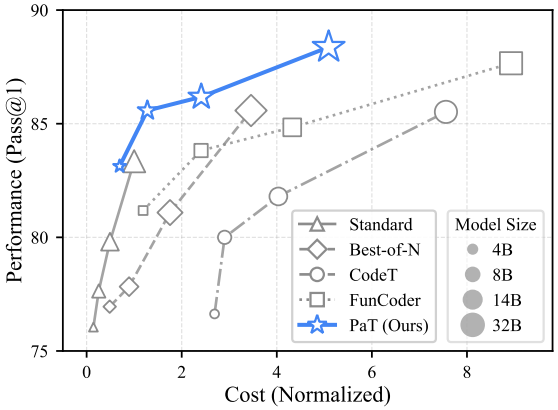


Figure 1: **Cost (↓) - Pass@1 (↑) trade-off across diverse sizes.** We plot the average Pass@1 across foundational benchmarks (HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021) and their EvalPlus (Liu et al., 2023) variants) against the relative inference cost. PaT consistently advances the Pareto frontier across model sizes ($\text{Qwen3}_{\text{4B,8B,14B and 32B}}$). Detailed results are provided in Section 4.1.1 and Table 1.

However, existing test-time scaling code generation methods often incur prohibitive inference cost. This is primarily due to their heavy reliance on explicit planning modules (Chen et al., 2024a; Le et al., 2024) or iterative repair and debugging loops (Shinn et al., 2023; Zhong et al., 2024), which consume significant computational resources. As illustrated in Figure 1, previous state-of-the-art methods such as FunCoder (Chen et al., 2024a) consume excessive test-time resources, often making a small model (Qwen3_{4B}) with this method more expensive than a much larger model ($\text{Qwen3}_{\text{32B}}$) with standard inference, even with inferior performance. This indicates that previous methods underperform standard scaling in their cost-performance ratio.

This inefficiency stems from ignoring the intrinsic difficulty distribution of the workload. As shown in Figure 1, a small model (Qwen3_{4B}) already achieves 76% Pass@1 using standard inference, suggesting that a substantial fraction of

*Corresponding author.

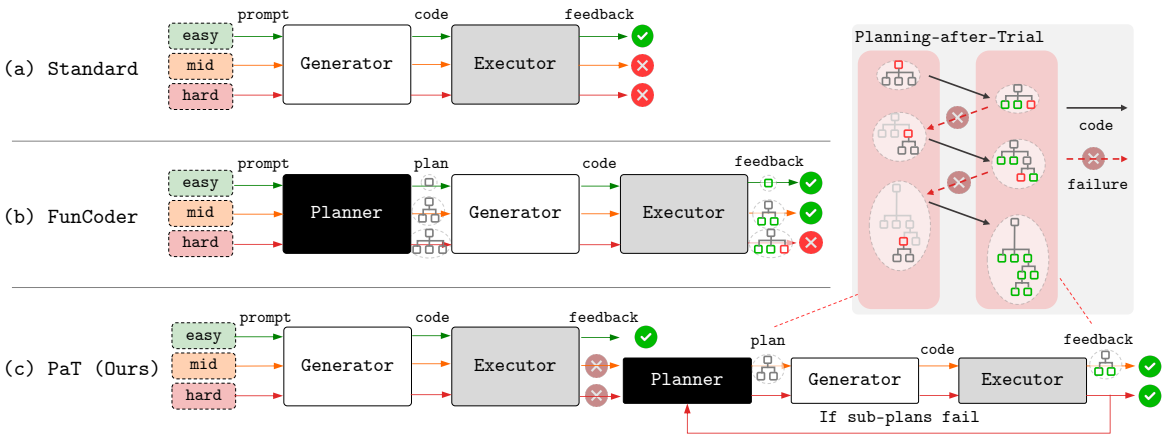


Figure 2: **Comparison with existing methods and PaT (Ours).** Problems are grouped by difficulty (easy, mid, and hard). Boxes denote the key components: a Generator (creates code), a Planner (decomposes the problem), and an Executor (verifies the solution). **(a) Standard** directly generate and execute; works on easy problems but often fails on harder ones. **(b) FunCoder (PbT)** always plans first, so planning cost is paid even when unnecessary. **(c) PaT (Ours)** trials first and plans only on failure; solves easy problems cheaply and hard problems adaptively.

tasks can be solved without complex interventions. Only a minority of hard instances truly require expensive planning. Nonetheless, many prior approaches (Chen et al., 2024a; Le et al., 2024; Jiang et al., 2024b) follow a rigid "Planning-before-Trial" (PbT) policy, wasting resources by indiscriminately applying expensive planning to these tractable problems.

To address this, we introduce "Planning-after-Trial" (PaT), an adaptive method that inverts PbT. Unlike PbT, which enforces planning for every problem, PaT initiates with standard inference and leverages execution feedback to verify this solution. As illustrated in Figure 2, expensive decomposition is triggered only when the initial attempt fails. By strictly limiting planning to problems that demonstrably require it, PaT concentrates computational resources precisely on hard instances while keeping simple ones cheap. Consequently, PaT significantly lowers the average inference cost while improving performance, thereby advancing the cost-performance Pareto frontier as shown in Figure 1.

The efficiency of PaT is further amplified within a heterogeneous model configuration. This enables a strategic division of labor, where a cost-efficient model is assigned to the high-volume "generator" role and a powerful one to the infrequent "planner" role. This alignment reflects the intuition that planning requiring more complex reasoning (*e.g.,* problem decomposition) is better handled by stronger models, while the resulting decomposed sub-tasks can often be implemented proficiently by smaller models.

We empirically validate our claims across diverse model families and a comprehensive suite of benchmarks. Our results first establish that PaT consistently outperforms the state-of-the-art PbT baseline, FunCoder (Chen et al., 2024a), achieving higher performance across all model scales while using, on average, only 60% of the inference cost. Furthermore, we show that the heterogeneous model configuration further enhances this efficiency. For instance, a small generator model guided by a powerful planner achieves competitive performance, with a <1% gap to a larger model in the homogeneous setting, while incurring only 31% of the cost. These findings confirm that PaT's adaptive policy establishes a new, superior cost-performance frontier. Collectively, our work demonstrates that scaling test-time compute need not be uniform, proving that unnecessary overhead can be effectively eliminated while maintaining or even enhancing performance.

Our contributions are summarized as follows:

- We propose Planning-after-Trial (PaT), an adaptive policy that invokes planning only upon verification failure to avoid unnecessary overhead.
- We pair PaT with a heterogeneous model configuration, combining small models for generation and large models for planning to maximize efficiency.
- We provide a comprehensive evaluation across model families and benchmarks, showing that our approach consistently advances the cost-performance Pareto frontier.

2 Related work

Code Generation with LLMs. To advance code generation, research has prioritized enhancing intrinsic capabilities via post-training (Dubey et al., 2024; Zhu et al., 2024) and self-improvement frameworks like rStar-Coder (Liu et al., 2025), SPIN (Chen et al., 2024b), and SCoRE (Kumar et al., 2024). In parallel, significant attention has shifted toward scaling test-time reasoning and computation given models. Key approaches utilize few-shot prompting (Brown et al., 2020) and diversity sampling (Chen et al., 2021), often refined by test-driven verification (Chen et al., 2022a; Shinn et al., 2023), iterative refinement (Wang et al., 2024; Zhang et al., 2024; Zhong et al., 2024), or retrieval from external bases (Li et al., 2022; Zhang et al., 2023). To tackle higher complexity, recent systems have further incorporated hierarchical decomposition (Chen et al., 2024a; Le et al., 2024) and natural language planning (Wang et al., 2025a). While these approaches improve robustness, they often apply sophisticated interventions uniformly across problems, incurring unnecessary overhead for simple cases. Notably, AdaCoder (Zhu et al., 2025) dynamically selects refinement strategies but still relies on iterative repair rather than structural decomposition. In contrast, PaT optimizes test-time computation, reserving complex interventions strictly for verified failures using execution feedback as a precise trigger.

Decomposition and Structured Reasoning. To tackle complex problems beyond the reach of direct generation, divide-and-conquer strategies have evolved from implicit reasoning steps in Chain-of-Thought (Wei et al., 2022) and Tree-of-Thoughts (Yao et al., 2023) to explicit planning paradigms. This evolution spans diverse domains, ranging from sequential solving in Least-to-Most prompting (Zhou et al., 2023) and mathematical reasoning with Program of Thoughts (Chen et al., 2022b), to using a society of models for complex tasks (Juneja et al., 2024). In the code domain, structured pipelines like Self-Plan (Jiang et al., 2024b) and FunCoder (Chen et al., 2024a) explicitly decompose problems before implementation. However, these approaches predominantly adhere to a rigid Planning-before-Trial (PbT) policy, where planning is invoked unconditionally prior to execution. While recent work attempts to mitigate this inefficiency by learning an adaptive policy for invoking planning (Paglieri et al., 2025), such approaches introduce additional training complexity and auxiliary models. In contrast, PaT offers a simpler, learning-free alternative: it utilizes the definitive execution signal as a reactive trigger, avoiding both the universal cost of PbT and the overhead of training separate policy models.

3 PaT: Planning after Trial

Problem Formulation. We model a code generation instance as a specification x (e.g., natural language description) and seek a program $\mathcal{F}$ that satisfies x. Let M_G denote a generator model and M_P a planner model. For any specification x, the generator M_G produces a direct candidate implementation, denoted $\hat{f}$. The planner M_P, when invoked, produces a decomposition plan consisting of a new top-level implementation, $\hat{f}$, and a set of subproblem specifications $\{x_i\}$. The final program $\mathcal{F}$ is constructed by COMPOSE that merges an implementation $\hat{f}$ with a set of verified helper functions H. For verification, we construct a test set $\mathcal{T}(x) = \{(\text{in}_j, \text{out}_j)\}_{j=1}^{t}$ and execute programs in a sandboxed Python runtime. We evaluate $\mathcal{F}$ with

$$\textsc{Evaluate}\big(\mathcal{F}, \mathcal{T}(x)\big) = \sum_{j=1}^{t} \mathbf{1}\big[\mathcal{F}(\text{in}_j) = \text{out}_j\big], \quad (1)$$

i.e., the number of tests passed.

3.1 Adaptive Planning via Failure Feedback

PaT executes a simple yet effective failure-triggered policy. Given the problem specification x, the generator M_G first attempts a Best-of-N trial by sampling multiple candidate solutions. We verify these candidates on a test set $\mathcal{T}(x)$. If any candidate passes all tests, the pipeline terminates and returns that solution immediately without incurring any decomposition overhead. However, if all candidates fail, this consistent failure serves as a strong signal that the problem complexity exceeds the generator's direct reasoning capability, rather than being a simple implementation error. Only in this case, PaT invokes the planner M_P to decompose x into subproblems $\{x_i\}$, which are solved recursively using the same trial-first policy.

When all subproblems have verified solutions, *i.e.,* $\textsc{Evaluate}(\hat{f}_i, \mathcal{T}(x_i)) = |\mathcal{T}(x_i)|$ for all i, PaT composes them into a parent-level candidate and re-verifies against $\mathcal{T}(x)$. If this composite solution passes, the process terminates successfully. Otherwise, the planner is invoked again on the original problem specification x. For this subsequent

Algorithm 1 PLANNING AFTER TRIAL (PAT)

Input: Problem x, Set of Helper Functions H,
  Generator M_G, Planner M_P
Output: Generated Program $\mathcal{F}$
PaT:
$\hat{f} \leftarrow M_G(x; H)$ ▷ Best-of-N trial
$\mathcal{T}(x) \leftarrow$ GENERATETESTS(x)
$p \leftarrow$ EVALUATE(COMPOSE$(\hat{f}, H), \mathcal{T}(x)$)
if $p = |\mathcal{T}(x)|$ **then**
  $\mathcal{F} \leftarrow$ COMPOSE$(\hat{f}, H)$
  return $\mathcal{F}$ ▷ Trial success
end if
while True **do** ▷ Until success or plateau
  $\hat{f}_{\text{prev}}, p_{\text{prev}} \leftarrow \hat{f}, p$
  $\hat{f}, \{x_i\} \leftarrow M_P(x; H)$ ▷ Planner invoked
  for each x_i not implemented in H **do**
    $\hat{f}_i \leftarrow$ PAT(x_i, H, M_G, M_P)
    $H \leftarrow H \cup \{\hat{f}_i\}$
  end for
  $p \leftarrow$ EVALUATE(COMPOSE$(\hat{f}, H), \mathcal{T}(x)$)
  if $p = |\mathcal{T}(x)|$ **then** ▷ Final success
    return COMPOSE$(\hat{f}, H)$
  end if
  if $p \leq p_{\text{prev}}$ **then** ▷ Plateau condition
    return COMPOSE$(\hat{f}_{\text{prev}}, H)$
  end if
end while

planning attempt, the planner is provided with the original problem and the set of previously successful subsolutions $\{\hat{f}_i\}$ as additional context. This allows the planner to make a more informed decomposition decision while reusing already successful components, further enhancing cost-efficiency.

3.2 Test Cases and Verification

Relying solely on provided public test cases is often insufficient. Benchmarks typically provide only basic scenarios that lack critical edge cases (Chen et al., 2021), or in some cases like MBPP, provide none at all (Austin et al., 2021). To ensure robust operation in such environments, we explicitly generate test cases $\mathcal{T}(x)$, producing an average of 6.7 test cases per problem.

However, prior works note that generated test cases can be noisy or incorrect (Chen et al., 2022a; Wang et al., 2025b; Prasad et al., 2025). To mitigate this, most previous works adopt consensus-based scoring to identify the most robust output (Chen et al., 2022a, 2024a). This approach evaluates a pool of candidate outputs against generated test

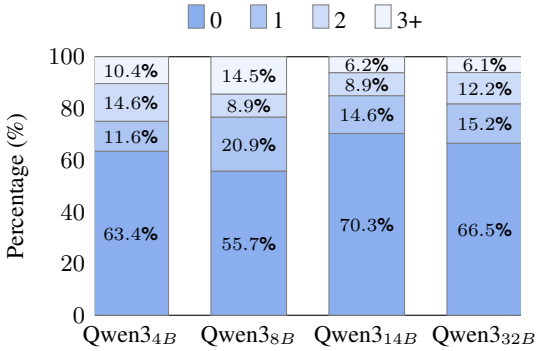


Figure 3: **Distribution of incorrect test cases (false positives) per problem on HumanEval.** The percentage of problems where the generated test cases contain 0, 1, 2, or 3+ incorrect test cases, given an average of 6.7 generated test cases per problem.

inputs, selecting the one that achieves the broadest consensus. However, such consensus-based scoring is ill-suited for our framework, as its objective of selecting the most likely correct outputs contrasts with PaT's need for a definitive binary signal (success or failure) to trigger its adaptive escalation to planning.

We therefore separate the success criterion from the termination rule. The success criterion is strict: a candidate is accepted only if it passes all test cases in $\mathcal{T}(x)$. This reliance on a strict signal is feasible because, as shown in Figure 3, generated test cases are completely error-free for the majority of problems (e.g., 63.4% on HumanEval with Qwen3$_{4B}$). Furthermore, even in noisy instances, errors are typically limited to 1-2 false positives, with severe noise (3+ errors) concentrated in a small minority. To handle these residual noisy test cases and prevent unproductive recursion, we track the pass count $p^{(t)} = \text{EVALUATE}(\hat{f}, \mathcal{T}(x))$ at iteration t and apply a plateau rule: if $p^{(t)} \leq p^{(t-1)}$ then halt and return the best-known solution. We find this heuristic to be practically effective, as it prevents the model from overfitting to potential false positives by terminating the search when performance saturates.

3.3 Heterogeneous Model Configuration

To further enhance efficiency, we adopt a heterogeneous model configuration that assigns distinct models to the two key roles inherent in PaT: the generator and the planner. These roles place different demands on model capability. The generator's task, which is to produce a solution for a well-scoped and often manageable subproblem, can be effectively handled by a cost-efficient small language

model (sLM). In contrast, the planner's task, which is to understand the nuances of a complex specification and propose a helpful decomposition, benefits from the advanced reasoning of a high-performance large language model (LLM). Building on this observation, we instantiate PaT with an sLM as the generator and an LLM as the planner.

However, achieving efficiency requires more than simply "using a smaller model as the generator." While decomposition is less expensive than full generation, it still incurs a planning token cost. We must therefore navigate a critical trade-off: an overly weak sLM triggers frequent failures, necessitating repeated interventions by the planner, which raises the total cost. Conversely, an overly strong sLM reduces the need for planning but increases the baseline cost of every trial, eroding the benefits of heterogeneity. Therefore, maximizing efficiency requires empirically tuning the generator selection to strike a balance between minimizing baseline costs and avoiding excessive planning overhead. To provide a rigorous foundation for these observations, we provide a formal analysis of this trade-off in Appendix A.

4 Experiments

In this section, we evaluate PaT in terms of both performance and cost efficiency. We conduct experiments on two settings: a homogeneous setting (Section 4.1), and a heterogeneous setting (Section 4.2). Below we describe the experimental setup used in our evaluation, including benchmarks, LLMs, evaluation metrics, and baselines.

Benchmarks. We evaluate on established code generation benchmarks. **HumanEval** (Chen et al., 2021) and **MBPP** (Austin et al., 2021) measure foundational code generation. For both, we use EvalPlus (Liu et al., 2023) to obtain expanded and more robust unit tests (*i.e.*, **HumanEval+** and **MBPP+**). For challenging tasks, we use **xCodeEval** (Khan et al., 2023) benchmark, partitioning instances into four categories (Easy, Mid, Hard, and Expert) following FunCoder's rating scheme.

LLMs. We primarily adopt the Qwen3 family (4B, 8B, 14B, and 32B) (Yang et al., 2025) as our main testbed, since it provides fine-grained scaling steps within a single model family. To evaluate the cross-family generalization of our method, we additionally conduct experiments on Llama-3.1-8B-Instruct (Dubey et al., 2024) and DeepSeek-Coder-V2-Lite-Instruct (≈16B) (Zhu et al., 2024). For the cost analysis, we calculate inference costs based on public market pricing as detailed in Appendix E.

Evaluation Metric. We report two primary metrics: **Pass@1** and **LLM cost**. Pass@1 is our primary performance metric, reflecting the ability to generate a correct solution on the first attempt, which aligns with real-world efficiency goals. LLM cost for each experiment is computed directly from the per-token prices in Table 7.

Baselines. To validate the effectiveness of PaT, we compare it against the following state-of-the-art algorithms for scaling test-time computation:

- **Standard** (Brown et al., 2020) uses few-shot prompting, giving several in-context examples to the model and asking it to generate a solution; we follow the original prompting protocol to measure baseline quality without post-hoc filtering or decomposition.
- **Best-of-N** (Chen et al., 2021) generates multiple candidates (N=5) and selects the best one; we include it to distinguish the gains of our adaptive planning from those achievable simply by scaling inference compute.
- **CodeT** (Chen et al., 2022a) performs consensus-based selection by executing solutions against automatically generated test cases; it serves as a representative benchmark for the efficacy of test-driven verification methods.
- **FunCoder** (Chen et al., 2024a) implements a static divide-and-conquer pipeline with a fixed hierarchy, solves subproblems, and assembles the final solution; we compare against it to highlight the efficiency gains of our adaptive, failure-driven policy over rigid Planning-before-Trial strategies.

For the sake of reproducibility and clarity, a detailed account of our implementation is included in Appendix C–G.

4.1 Homogeneous Setting

In this section, we compare results from running PaT using a homogeneous language model against baselines. Evaluations are performed on the foundational benchmarks (HumanEval and MBPP; Sec. 4.1.1) and the challenging benchmark (xCodeEval; Sec. 4.1.2).

Model	Method	HumanEval		MBPP		Avg.	Δ Avg.	Cost
		base	plus	base	plus			
$Qwen3_{4B}$	Standard	78.66	70.12	79.43	76.00	76.05	-	1.00
	Best-of-N	79.27	71.95	80.00	76.57	76.95	+ 0.90	3.39
	CodeT	78.66	70.73	80.00	77.14	76.63	+ 0.58	18.82
	FunCoder	85.98	79.88	81.14	77.71	81.18	+ 5.13	8.31
	PaT (Ours)	**89.63**	**82.32**	**82.29**	**78.29**	**83.13**	+ 7.08	4.85
$Qwen3_{8B}$	Standard	78.66	71.34	81.14	79.43	77.64	-	1.00
	Best-of-N	80.49	72.56	80.57	77.71	77.83	+ 0.19	3.50
	CodeT	80.49	73.78	84.57	81.14	80.00	+ 2.36	11.37
	FunCoder	89.02	81.10	83.43	81.71	83.82	+ 6.18	9.43
	PaT (Ours)	**90.85**	**82.32**	**85.14**	**84.00**	**85.58**	+ 7.94	5.00
$Qwen3_{14B}$	Standard	83.54	76.83	80.57	78.29	79.81	-	1.00
	Best-of-N	82.93	77.44	82.86	81.14	81.09	+ 1.28	3.58
	CodeT	83.54	76.22	**85.71**	81.71	81.80	+ 1.99	8.22
	FunCoder	89.63	81.71	85.14	82.86	84.84	+ 5.03	8.82
	PaT (Ours)	**91.46**	**83.54**	**85.71**	**84.00**	**86.18**	+ 6.37	4.91
$Qwen3_{32B}$	Standard	87.20	80.49	83.43	82.29	83.35	-	1.00
	Best-of-N	90.24	82.93	85.14	84.00	85.58	+ 2.23	3.46
	CodeT	89.02	80.49	86.86	85.71	85.52	+ 2.17	7.56
	FunCoder	93.29	**84.76**	86.86	85.71	87.66	+ 4.31	8.93
	PaT (Ours)	**93.90**	84.15	**88.57**	**86.86**	**88.37**	+ 5.02	5.09
$Llama3.1_{8B}$	Standard	68.29	59.15	66.86	66.29	65.15	-	1.00
	Best-of-N	70.12	62.20	72.57	69.14	68.51	+ 3.36	2.60
	CodeT	71.95	61.59	72.00	71.43	69.24	+ 4.09	4.86
	FunCoder	75.00	67.68	72.00	71.43	71.53	+ 6.38	8.07
	PaT (Ours)	**78.05**	**68.90**	**74.29**	**72.00**	**73.31**	+ 8.16	5.32
DeepSeek-Coder	Standard	83.54	75.61	80.57	78.86	79.65	-	1.00
	Best-of-N	81.10	75.00	81.71	81.14	79.74	+ 0.09	3.05
	CodeT	81.10	72.56	84.57	82.86	80.27	+ 0.62	6.40
	FunCoder	85.98	79.27	**85.14**	84.00	83.60	+ 3.95	8.77
	PaT (Ours)	**86.59**	**79.88**	**85.14**	**85.14**	**84.19**	+ 4.54	5.97

Table 1: **Performance and cost comparison on foundational benchmarks.** We report Pass@1 across four benchmarks (HumanEval, HumanEval+, MBPP, and MBPP+) and their average ('Avg.'). Cost is normalized relative to the Standard baseline (1.00). Best results are in **bold**.

4.1.1 Foundational Benchmarks

We evaluate PaT and baselines on the foundational benchmarks (HumanEval and MBPP) and their EvalPlus variants. As shown in Table 1, PaT consistently outperforms all baselines across all tested model families and scales. The effectiveness of this policy is best illustrated by its ability to dramatically improve the capability of smaller models. For instance, PaT enables $Qwen3_{4B}$ to achieve an average Pass@1 of 83.13%, which is remarkably similar to the 83.35% achieved by Standard on $Qwen3_{32B}$, a model eight times larger. This demonstrates that PaT is not merely an incremental improvement but a powerful policy that fundamentally alters the performance curve of a given model.

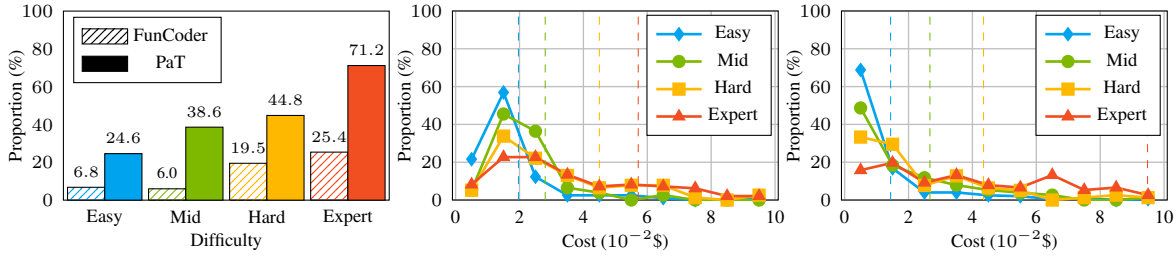
Beyond its superior performance, PaT also shows critical advantages in cost-efficiency. As detailed in Table 1, PaT consistently achieves higher performance than FunCoder, the previous state-of-the-art hierarchical method, while consuming, on average 60% of its cost. This efficiency gain can be explained by examining Standard performance. On these foundational benchmarks, Standard solves, on average, 76% of problems directly, demonstrating that a large majority of tasks do not require decomposition. FunCoder's rigid PbT policy is forced to pay a cost of planning on all of these problems, incurring its expensive planning overhead even when it is not needed. In contrast, PaT only invokes its planner on the much smaller fraction of problems (24%) that actually fail the initial, cheaper trial. By avoiding this universal planning overhead, PaT establishes a new and more effective cost-performance frontier for code generation.

4.1.2 Challenging Benchmark

On the challenging xCodeEval benchmark, PaT again achieves higher performance than all baselines across all model variants. However, we observe an interesting cost dynamic: for smaller models like $Qwen3_{4B}$ and $Llama3.1_{8B}$, PaT incurs a

Model	Method	Easy	Mid	Hard	Expert	All	Δ All	Cost
$Qwen3_{4B}$	Standard	37.70	17.86	3.45	0.00	18.40	-	1.00
	Best-of-N	51.91	29.46	8.05	0.00	27.00	+ 8.60	4.06
	CodeT	40.44	20.54	3.45	0.00	20.00	+ 1.60	21.64
	FunCoder	55.19	29.46	12.64	0.00	29.00	+ 10.60	12.95
	PaT (Ours)	**61.75**	**40.18**	**14.94**	0.00	**34.20**	+ 16.20	17.93
$Qwen3_{8B}$	Standard	54.10	28.57	5.75	0.00	27.20	-	1.00
	Best-of-N	66.67	42.86	6.90	0.00	35.20	+ 8.00	3.34
	CodeT	45.92	31.25	8.05	0.00	25.20	- 2.00	17.00
	FunCoder	64.48	43.75	9.20	0.00	35.00	+ 7.80	8.62
	PaT (Ours)	**69.95**	**45.54**	**11.49**	0.00	**37.80**	+ 10.60	6.98
$Qwen3_{14B}$	Standard	53.55	36.61	9.20	0.00	25.20	-	1.00
	Best-of-N	68.31	50.00	13.79	0.00	38.60	+ 13.40	3.16
	CodeT	55.19	41.07	9.20	0.00	31.00	+ 5.80	7.48
	FunCoder	73.22	52.68	18.39	0.00	41.80	+ 16.60	9.03
	PaT (Ours)	**73.77**	**53.57**	**21.84**	**0.85**	**43.00**	+ 17.80	6.49
$Qwen3_{32B}$	Standard	54.64	39.29	11.49	0.00	30.80	-	1.00
	Best-of-N	71.04	50.00	14.94	0.00	39.80	+ 9.00	3.28
	CodeT	57.92	39.29	12.64	0.00	32.20	+ 1.40	7.55
	FunCoder	**74.86**	**54.46**	16.09	0.00	42.40	+ 11.60	7.87
	PaT (Ours)	74.32	**54.46**	**18.39**	**1.69**	**43.00**	+ 12.20	6.00
$Llama3.1_{8B}$	Standard	13.11	3.57	1.15	0.00	5.80	-	1.00
	Best-of-N	22.40	8.04	1.15	0.00	10.20	+ 4.40	1.39
	CodeT	13.11	3.57	1.15	0.00	5.80	+ 0.00	2.59
	FunCoder	26.78	7.14	**2.30**	0.00	11.80	+ 6.00	5.46
	PaT (Ours)	**32.79**	**11.61**	**2.30**	0.00	**15.00**	+ 14.20	8.42
DeepSeek-Coder	Standard	43.17	19.64	9.20	0.85	22.00	-	1.00
	Best-of-N	59.56	26.79	10.34	0.85	29.80	+ 7.80	2.57
	CodeT	46.99	22.32	9.20	0.00	23.80	+ 1.80	5.46
	FunCoder	62.30	31.25	12.64	**1.69**	32.40	+ 10.40	8.69
	PaT (Ours)	**63.39**	**33.93**	**13.79**	**1.69**	**33.60**	+ 11.60	7.69

Table 2: **Performance breakdown on xCodeEval by difficulty.** We report Pass@1 scores for each difficulty category (Easy, Mid, Hard, and Expert) and the overall score.



(a) Decomposition rate (b) Cost (FunCoder) (c) Cost (PaT)

Figure 4: **Adaptive decomposition probability and cost analysis for $Qwen3_{4B}$ on xCodeEval.** (a) Decomposition rate of FunCoder and PaT by problem difficulty. Per-difficulty cost distribution (solid line) and average cost (vertical dashed line) on (b) FunCoder and (c) PaT.

higher cost than FunCoder. This is not a sign of inefficiency but a direct consequence of PaT's adaptive strategy. Less capable models fail more frequently on xCodeEval's difficult problems, and PaT correctly interprets these failures as signals to invest in decomposition. This strategic escalation, while costly, is precisely what allows these smaller models to overcome challenges where rigid policies like FunCoder's fall short.

For a deeper analysis, we examine the behavior of $Qwen3_{4B}$ in Figure 4, which reveals a fundamental limitation of pre-emptive planning. While both PaT and FunCoder increase decomposition for harder problems, PaT exhibits a much more dynamic response compared to FunCoder's more restrained increase. Remarkably, despite this aggressive decomposition, the cost difference remains marginal because FunCoder incurs a planning overhead on 100% of problems, whereas PaT invokes the planner exclusively on failed instances. This reinforces a key insight: planning is a high-cost resource that must be allocated reactively, relying on the signal of failure to distinguish necessary investment from wasteful overhead.

```
...
- If a single function is too hard to solve,
    you can decompose it into multiple
    smaller functions.
...
```

(a) Decomposition prompt for FunCoder

```
...
- The previous attempt to direct implement
    the target function is failed, indicating
     its overall logic might be too complex
    to implement directly.
- Therefore, you must decompose it into
    multiple smaller, manageable helper
    functions.
...
```

(b) Planning prompt for PaT

Figure 5: **Comparison of planning prompts.** Excerpts from planning prompts for (a) FunCoder and (b) PaT.

This behavioral divergence stems from the distinct prompt strategies shown in Figure 5. While FunCoder relies on the model's subjective assessment of complexity, PaT leverages a grounded feedback signal derived from trial failure to issue an imperative command. This explicit directive proves far more effective at triggering necessary decomposition than a conditional instruction, ensuring the model does not underestimate the task's difficulty.

4.2 Heterogeneous Setting

Our discussion in Section 3.3 established the potential for enhanced cost-efficiency in a heterogeneous model configuration. To empirically validate this, we conduct experiments on the foundational benchmarks, pairing a powerful planner (Qwen3_{32B}) with a series of smaller generator models. The results, presented in Table 3, provide strong support for our analysis. Pairing a Qwen3_{8B} generator with a Qwen3_{32B} planner achieves competitive performance (an average Pass@1 of 87.39%), with a <1% gap to the homogeneous Qwen3_{32B}, while reducing the relative cost to just 0.31.

The superior cost-benefit trade-off of the heterogeneous approach is visualized in Figure 6. In this graph, the slope of the curve represents the performance return on additional cost. The heterogeneous model configurations exhibit a significantly steeper slope, demonstrating that upgrading only the planner is a highly capital-efficient strategy, yielding substantial performance gains for a marginal increase in cost. This confirms our central hypothesis: because PaT invokes the planner infrequently, reserving a powerful model for this critical but rare task is the most cost-effective way to enhance the overall system's capability.

Generator	Planner	Avg.	Cost
Qwen3_{32B}	Qwen3_{32B}	88.37	1.00
Qwen3_{14B}	Qwen3_{14B}	86.18	0.47
	Qwen3_{32B}	87.53	0.49
Qwen3_{8B}	Qwen3_{8B}	85.58	0.25
	Qwen3_{32B}	87.39	0.31
Qwen3_{4B}	Qwen3_{4B}	83.13	0.14
	Qwen3_{32B}	84.78	0.18

Table 3: **Performance (Pass@1) and relative cost results** for homogeneous and heterogeneous PaT configurations. The cost is normalized relative to the homogeneous Qwen3_{32B} model, which is set to 1.00.

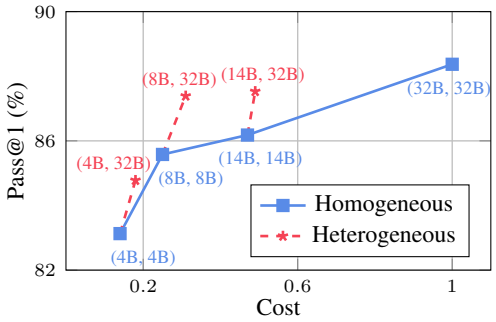


Figure 6: **Trade-off curve for heterogeneous configurations.** Corresponding to Table 3, this plot visualizes the Pass@1 performance vs. relative cost. Labels denote (Generator, Planner) sizes. Employing a powerful Qwen3_{32B} planner (dashed line) yields a significant performance gain for a marginal increase in cost compared to Homogeneous configurations (solid line).

5 Conclusion

In this work, we introduced Planning-after-Trial (PaT), an adaptive policy that addresses the critical challenge of high inference costs in LLM-based code generation. By inverting the conventional Planning-before-Trial (PbT) policy, PaT avoids unnecessary planning overhead on simple problems and strategically allocates computational resources only when a direct attempt fails. We demonstrated that this adaptive nature is uniquely synergistic with a heterogeneous configuration, allowing for the decoupling of planning and generation costs. Comprehensive experiments confirmed that PaT outperforms existing state-of-the-art baselines in homogeneous settings across diverse models and benchmarks. Furthermore, we validated that our heterogeneous model configuration establishes a superior cost-performance frontier compared to homogeneous setups. By enabling principled test-time computation scaling, PaT provides a practical and effective framework for building more scalable and cost-efficient code generation systems.

Limitations

Relying on self-generated test cases inherently introduces noise and potential inefficiency arising from unnecessary planning for correctly solved problems. While integrating emerging high-fidelity generation methods (Wang et al., 2025b; Prasad et al., 2025) offers synergistic potential, achieving a perfectly noise-free oracle remains unlikely; thus, robust mechanisms like our plateau heuristic remain essential for practical deployment. Regarding computational resources, our heterogeneous configuration entails a higher static memory footprint. However, this design aligns with efficient architectures like Mixture-of-Experts (MoE) (Jiang et al., 2024a), prioritizing the significant reduction of average dynamic inference cost and user-facing latency for the majority of tasks.

Acknowledgements

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea government (MSIT) (RS-2019-II191906, Artificial Intelligence Graduate School Program (POSTECH); RS-2024-00457882, AI Research Hub Project; IITP-2026-RS-2024-00437866, Information Technology Research Center (ITRC); RS-2026-25511821, Development of Personalized Media Service Recommendation and Generative Technology). It was also supported by the MSIT under the Global Research Support Program in the Digital Field (RS-2024-00436680) supervised by the IITP. This project is supported by Microsoft Research Asia.

AI writing assistance

We utilized large language models solely for the purpose of refining the clarity, grammar, and style of the manuscript under the scope of "Assistance purely with the language of the paper." All scientific claims, experimental designs, and empirical results presented in this paper are the original work of the authors.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022a. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Jingchang Chen, Hongxuan Tang, Zheng Chu, Qianglong Chen, Zekun Wang, Ming Liu, and Bing Qin. 2024a. Divide-and-conquer meets consensus: Unleashing the power of functions in code generation. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022b. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024b. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, and 1 others. 2024a. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Gurusha Juneja, Subhabrata Dutta, and Tanmoy Chakraborty. 2024. $\texttt{LM}^2$: A simple society of language models solves complex reasoning. *arXiv preprint arXiv:2404.02255*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2023. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, and 1 others. 2024. Training language models to self-correct via reinforcement learning. *arXiv preprint arXiv:2409.12917*.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. In *ICLR*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, R'emi 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.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Yifei Liu, Li Lyna Zhang, Yi Zhu, Bingcheng Dong, Xudong Zhou, Ning Shang, Fan Yang, and Mao Yang. 2025. rstar-coder: Scaling competitive code reasoning with a large-scale verified dataset. *arXiv preprint arXiv:2505.21297*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

OpenAI. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Davide Paglieri, Bartłomiej Cupiał, Jonathan Cook, Ulyana Piterbarg, Jens Tuyls, Edward Grefenstette, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. 2025. Learning when to plan: Efficiently allocating test-time compute for llm agents. *arXiv preprint arXiv:2509.03581*.

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. Learning to generate unit tests for automated debugging. *arXiv preprint arXiv:2502.01619*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2025. Scaling llm test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*.

Evan Z Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, William Song, Vaskar Nath, Ziwen Han, Sean M Hendryx, Summer Yue, and Hugh Zhang. 2025a. Planning in natural language improves llm search for code generation. In *The Thirteenth International Conference on Learning Representations*.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. Intervenor: Prompting the coding ability of large language models with the interactive chain of repair. In *ACL (Findings)*.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025b. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *ASE*.

Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 851–870.

Denny Zhou, Nathanael Sch"arli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and 1 others. 2023. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Yueheng Zhu, Chao Liu, Xuan He, Xiaoxue Ren, Zhongxin Liu, Ruwei Pan, and Hongyu Zhang. 2025. Adacoder: An adaptive planning and multi-agent framework for function-level code generation. *arXiv preprint arXiv:2504.04220*.

A Theoretical Analysis

In this section, we provide a formal theoretical analysis supporting the cost-efficiency of the Heterogeneous Model Configuration in Section 3.3. We first establish the notations and cost model, then prove the existence of an efficient heterogeneous policy (Theorem 1). We subsequently extend our analysis to consider the asymptotic cost for problems of large complexity k and derive the optimal cost for the generator sLM under scaling laws.

Preliminaries and Notations. We denote the problem complexity by $k > 0$. We consider a set of models $M \in \{L, s\}$, where L represents a large, capable model (used as the Planner) and s represents a smaller, cost-efficient model (used as the Generator). For each model M, let p_M denote its problem-solving capability (the maximum complexity it can solve in a single attempt) and c_M denote its unit inference cost. Consistent with the heterogeneous setting, we assume $0 < p_s < p_L$ and $0 < c_s < c_L$.

Based on these notations, we define the core assumptions that characterize our cost model.

Assumption 1 (Generation cost)**.** *A generation call by M_G on a problem of complexity k incurs*

$$\mathit{Cost}_{\mathit{Generation}} = \begin{cases} k\, c_{M_G}, & k \le p_{M_G} \text{ (success)}, \\ p_{M_G}\, c_{M_G}, & k > p_{M_G} \text{ (failed)}. \end{cases} \tag{2}$$

Assumption 2 (Planning cost)**.** *A Planner-produced n-way plan maps a problem of size k to n independent subproblems of size $\frac{k}{n}$ and incurs a fixed per-subproblem overhead D_{M_P}, so that*

$$\mathit{Cost}_{\mathit{Planning}} = nD_{M_P}. \tag{3}$$

To formally connect a model's capability p_M and its unit cost c_M, we adopt a standard assumption from the literature on neural scaling laws.

Assumption 3 (Scaling law)**.** *We assume that a model's capability p_M and its unit cost c_M are related by a power-law scaling relation, consistent with prior work (Kaplan et al., 2020):*

$$p_M = \alpha\, c_M^{\beta}, \qquad \alpha > 0,\ 1 \ge \beta > 0. \tag{4}$$

The constraint $0 < \beta \le 1$ reflects the principle of diminishing returns.

With these definitions established, we now formally state the condition under which the heterogeneous policy outperforms the homogeneous one.

Theorem 1 (Existence of an efficient sLM capability)**.** *Let $k \sim \mathrm{Unif}(0, p_L]$. Under Assumptions 1, 2, and 3, if the total planning overhead satisfies*

$$nD_L < \left(\tfrac{1}{2} - \tfrac{1}{n^2}\right) p_L c_L, \tag{5}$$

then there exists $p_s \in (0, p_L)$ for which PaT with a Heterogeneous Model Configuration has strictly lower expected cost than an LLM-only policy.

Proof. First, for the homogeneous LLM-only policy, the cost for any problem $k \le p_L$ is kc_L. The expected cost is therefore:

$$\begin{aligned} &\mathbb{E}[\mathrm{Cost}_{\mathrm{Homogeneous}}] \\ &= \frac{1}{p_L}\int_0^{p_L} kc_L\, dk = \frac{1}{p_L}\frac{p_L^2 c_L}{2} = \frac{p_L c_L}{2} \end{aligned} \tag{6}$$

Next, for the Heterogeneous policy, we consider two cases based on the capability of the small model, p_s. If $p_s \ge k$, the generator s succeeds, incurring a cost of kc_s. Else, *i.e.,* $p_s < k$, the generator s fails (cost $p_s c_s$), the planner L is invoked (cost nD_L), and then the generator s solves the n subproblems of size $\frac{k}{n}$. Consider $p_s \ge \frac{p_L}{n}$, we can calculate the cost in this case $p_s c_s + nD_L + kc_s$. The expected cost is the sum of the integrals over

these two ranges, divided by p_L:

$$\mathbb{E}[\text{Cost}_{\text{Heterogeneous}}]$$
$$= \frac{1}{p_L}\left(\int_0^{p_s} kc_s\, dk + \int_{p_s}^{p_L} p_s c_s + nD_L + kc_s\, dk\right) \tag{7}$$
$$= \frac{1}{p_L}\left(\frac{p_s^2 c_s}{2} + p_s c_s(p_L - p_s) + nD_L(p_L - p_s) + \frac{(p_L^2 - p_s^2)c_s}{2}\right) \tag{8}$$
$$= \frac{p_L c_s}{2} + \frac{p_L - p_s}{p_L}(p_s c_s + nD_L) \tag{9}$$

To analyze a concrete scenario, let's assume we can choose an sLM such that its capability is a fraction of the LLM's, *i.e.,* $p_s = \frac{p_L}{n}$. Using the scaling law from Assumption 3 ($p_M = \alpha c_M^{\beta}$), we can relate the costs: $c_s = \left(\frac{p_s}{\alpha}\right)^{1/\beta} = \left(\frac{p_L}{n\alpha}\right)^{1/\beta} = n^{-1/\beta} c_L$.

Substituting these into the heterogeneous cost equation 9:

$$\mathbb{E}[\text{Cost}_{\text{Heterogeneous}}] \tag{10}$$
$$= \frac{p_L c_s}{2} + \frac{p_L - p_s}{p_L}(p_s c_s + nD_L) \tag{11}$$
$$= \frac{p_L n^{-1/\beta} c_L}{2} + \frac{n-1}{n}\left(\frac{p_L}{n} n^{-1/\beta} c_L + nD_L\right) \tag{12}$$
$$= \left(\frac{1}{2} + \frac{n-1}{n^2}\right) n^{-1/\beta} p_L c_L + (n-1)D_L \tag{13}$$

For the heterogeneous policy to be strictly more efficient, the difference must be positive:

$$0 > \mathbb{E}[\text{Cost}_{\text{Heterogeneous}} - \text{Cost}_{\text{Homogeneous}}] \tag{14}$$
$$= (n-1)D_L + \left(\frac{1}{2} + \frac{n-1}{n^2}\right) n^{-\frac{1}{\beta}} p_L c_L - \frac{p_L c_L}{2}. \tag{15}$$

Rearranging to solve for the planning overhead nD gives the condition stated in the theorem:

$$nD_L < \frac{n}{n-1}\left(\frac{p_L c_L}{2} - \left(\frac{1}{2} + \frac{n-1}{n^2}\right) n^{-\frac{1}{\beta}} p_L c_L\right) \tag{16}$$
$$= \left(\frac{n - n^{1-\frac{1}{\beta}}}{2(n-1)} - n^{-1-\frac{1}{\beta}}\right) p_L c_L. \tag{17}$$

As a specific intuitive case, if we consider linear scaling where $\beta = 1$, the condition simplifies to:

$$nD_L < \left(\frac{1}{2} - \frac{1}{n^2}\right) p_L c_L \tag{18}$$

This shows that as long as the total planning cost is less than the savings achieved by the heterogeneous configuration, a more efficient sLM exists. □

Theorem 2 (Asymptotic efficiency of the heterogeneous configuration)**.** *For any sufficiently complex task k, the Heterogeneous strategy is asymptotically more cost-efficient than the Homogeneous strategy, provided that the cost of decomposition satisfies the following condition:*

$$D_L < \frac{c_L - c_s}{\frac{1}{p_s} - \frac{1}{p_L}}. \tag{19}$$

Proof. We prove the theorem by analyzing the asymptotic cost of each strategy for a large problem of complexity k. Let $h_M = \lceil k/p_M \rceil$ be the number of recursive division levels for a model M. The total number of division operations is $1 + n + n^2 + ... + n^{h_M - 1} = \frac{n^{h_M}-1}{n-1}$. For a large k, we can approximate it as $\frac{n^{h_M}-1}{n-1} \approx \frac{k}{p_M(n-1)}$.

The total cost for a model M is the sum of three components: the cumulative cost of failures at each division step $\frac{k}{p_M(n-1)} p_M c_M$, the cumulative cost of decomposition $\frac{k}{p_M(n-1)} nD_L$, and the final cost of conquering the sub-problems kc_M. Summing these components gives the total asymptotic cost:

$$\frac{nk}{n-1}\left(c_M + \frac{D_L}{p_M}\right). \tag{20}$$

For the heterogeneous setting to be more cost-efficient than the homogeneous one, it requires:

$$\frac{nk}{n-1}\left(c_s + \frac{D_L}{p_s}\right) < \frac{nk}{n-1}\left(c_L + \frac{D_L}{p_L}\right). \tag{21}$$

The $\frac{nk}{n-1}$ term cancels. Rearranging the remaining terms to solve for D_L yields the condition stated in the theorem:

$$D_L < \frac{c_L - c_s}{\frac{1}{p_s} - \frac{1}{p_L}}. \tag{22}$$

□

This theorem provides a theoretical basis for the efficiency of heterogeneous model configuration. It indicates that the heterogeneous configuration becomes more cost-efficient when the cost of decomposition (D_L) is less than the savings generated by executing the sub-problems with a more cost-effective model (s). This provides a formal

Generator	Planner	HumanEval		MBPP		Avg.	Δ Avg.	cost
		base	plus	base	plus			
$Qwen3_{32B}$	$Qwen3_{32B}$	93.90	84.15	88.57	86.86	88.37	-	1.00
$Qwen3_{14B}$	$Qwen3_{14B}$	91.46	83.54	85.14	82.29	86.18	- 2.19	0.47
	$Qwen3_{32B}$	93.29	85.98	86.29	84.57	87.53	- 0.84	0.49
$Qwen3_{8B}$	$Qwen3_{8B}$	90.85	82.32	85.14	84.00	85.58	- 2.79	0.25
	$Qwen3_{32B}$	93.90	85.37	86.29	84.00	87.39	- 0.98	0.31
$Qwen3_{4B}$	$Qwen3_{4B}$	89.63	82.32	82.29	78.29	83.13	- 5.24	0.14
	$Qwen3_{32B}$	92.07	84.76	82.29	80.00	84.78	- 3.40	0.18

Table 4: **Full results on heterogeneous setting.** We report the average cost relative to $Qwen3_{32B}$ is normalized to 1.00.

rationale for using the heterogeneous configuration and shows that the PaT policy is a structured approach to allocating computational resources.

Theorem 3 (Optimal Generator cost under scaling laws)**.** *The cost of the optimal small model, c_s^*, that minimizes the asymptotic cost of the heterogeneous configuration is given by the following closed-form solution:*

$$c_s^* = \min\left\{\left(\frac{\beta \cdot D_L}{\alpha}\right)^{\frac{1}{\beta+1}}, c_L\right\} \tag{23}$$

Proof. The asymptotic cost of the heterogeneous configuration is proportional to the coefficient $c_s + \frac{D_L}{p_s}$. Substituting the scaling law gives $c_s + \frac{D_L}{\alpha c_s^\beta}$. To find the minimum, we take the derivative with respect to c_s and set it to zero:

$$1 - \frac{\beta D_L}{\alpha} c_s^{-\beta-1} = 0. \tag{24}$$

Solving for c_s yields the unconstrained optimum. The final solution is capped at c_L to respect the problem's practical constraints. □

This theorem provides a practical, closed-form solution for the cost of the optimal generator model in the asymptotic regime. The result serves as a powerful heuristic to approximately estimate the most cost-effective smaller model to use in real-world heterogeneous configurations.

Empirical Validation. To validate the practical utility of Theorem 3, we applied the formula to our HumanEval experimental data using Qwen3 models. We modeled model capability p against normalized input token costs $c \in \{0.11, 0.18, 0.35, 0.70\}$ using the scaling law $p = \alpha c^\beta$. Fitting this to our observed data yielded $\alpha \approx 1722.2$ and $\beta \approx 0.12$, with an average planning cost $D_L \approx 1270.73$ derived from the heterogeneous (4B+32B) experiments. Substituting these parameters into Theorem 3 yields a theoretical optimal cost $c_s^* \approx 0.114$. This value is remarkably close to the actual cost of the 4B model ($c = 0.11$). While our main experiments (Figure 6) suggest the 8B model offers a strong performance-cost balance, the 4B model is indeed the strictly cost-optimal generator as predicted. This alignment confirms that despite simplified assumptions, our theoretical model successfully captures the underlying cost dynamics and serves as a practical guideline for selecting the initial sLM size for hyperparameter search.

B Additional Results

B.1 Full version of Table 3

Table 4 presents the full results for our heterogeneous setting experiments across all four foundational benchmarks (HumanEval, HumanEval+, MBPP, and MBPP+). In this configuration, we pair a series of smaller generator models with a powerful, fixed planner ($Qwen3_{32B}$). The table provides the detailed Pass@1 scores and relative costs for each combination, which were aggregated and visualized in Figure 6 in the main body of the paper.

B.2 Wall-Clock Latency Analysis

To complement the token-cost analysis in the main paper, we measure end-to-end wall-clock latency in an isolated single-GPU environment to avoid interference from shared resources. Table 5 reports the mean per-problem latency (in seconds) on HumanEval using Qwen3 models.

PaT incurs roughly 3× overhead over Standard, while FunCoder incurs 5×, confirming that PaT's adaptive policy reduces not only token cost but also real-world execution time. Notably, the het-

Model	Standard	FunCoder	PaT
Qwen3_{4B}	3.87	19.82	11.14
Qwen3_{32B}	14.21	63.56	41.39
Qwen3_{4B+32B}	-	-	11.90

Table 5: **Mean wall-clock latency per problem (seconds) on HumanEval.** Measured in an isolated environment using Qwen3 models. Qwen3_{4B+32B} denotes the heterogeneous model configuration using a 4B generator and 32B planner.

Model	Best-of-N	CodeT	FunCoder	PaT
Qwen3_{4B}	0.38	0.03	0.70	**1.84**
Qwen3_{8B}	0.08	0.23	0.73	**1.99**
Qwen3_{14B}	0.50	0.28	0.64	**1.63**
Qwen3_{32B}	0.91	0.33	0.54	**1.23**
Llama3.1_{8B}	**2.10**	1.06	0.90	1.89
DeepSeek-Coder	0.04	0.11	0.51	**0.91**
Average	0.67	0.34	0.67	**1.58**

Table 6: **Compute ROI across models and methods on foundational benchmarks.** ROI is defined as $\Delta\text{Avg}/(\text{Cost}-1)$, measuring Pass@1 improvement per unit of additional cost over Standard. Higher is better.

erogeneous configuration (4B+32B) achieves a latency of 11.90s, nearly identical to the 4B-only PaT (11.14s) and faster than the 32B Standard baseline (14.21s). This demonstrates that reserving the larger model exclusively for planning keeps wall-clock time dominated by the cheaper generator.

B.3 Compute Return on Investment

To provide a unified metric for the cost-performance trade-off, we define the compute return on investment (ROI) as the performance gain per unit of additional cost over Standard:

$$\text{ROI} = \frac{\Delta\text{Avg}}{\text{Cost} - 1}, \tag{25}$$

where ΔAvg is the average Pass@1 improvement over Standard across benchmarks, and Cost is the normalized cost reported in Table 1 (Standard = 1). A higher ROI indicates a more efficient use of additional compute. Table 6 reports the ROI for all methods.

PaT achieves an average ROI of 1.58, approximately 2.4× higher than FunCoder (0.67) and 4.6× higher than CodeT (0.34). This confirms that PaT's adaptive policy consistently delivers superior returns on additional test-time investment across all model scales and families.

C Implementation Details

Generation. The trial phase of our PaT policy incorporates a Best-of-N strategy to maximize the chance of a direct success. For each problem specification, the generator model M_G produces 5 candidate solutions, using a temperature of 0.8 to encourage diverse outputs. Each of these 5 candidates is then verified against the test set. If any of the candidates passes all test cases, the process terminates successfully, and that solution is returned. Only if all 5 candidates fail the verification step does the policy escalate to the planning phase. The generation is retried up to 3 times if it fails to produce a parsable output.

Planning. If the trial fails, the planning phase is triggered. The planner model M_P is prompted to decompose the problem specification x by rewriting the main function to call new unimplemented helper functions. A single decomposition plan is generated with a temperature of 0.2. From the resulting Python code block, we parse the new function signatures and docstrings, which become the specifications for the subproblems. If the output is not a valid code block, this planning step is retried up to 3 times, and the maximum recursion depth is limited to 3 to prevent overly complex solutions. Once all subproblems are recursively solved, their solutions are composed into a final program and verified against the original test suite for x. If this composite solution still fails, PaT initiates a re-planning loop. The planner is invoked again on the original problem x, but this time it is provided with the set of previously successful helper functions as additional context, enabling a more informed and cost-efficient re-planning attempt.

Test case generation & verification. Our verification process requires a test suite $\mathcal{T}(x)$ for each problem x, which we construct in two stages. First, we process any example test cases provided directly in the problem description. Then, to ensure a comprehensive and robust evaluation, we augment the initial suite by prompting an LLM to generate additional test cases based on the problem description, a technique inspired by CodeT. This test generation process is performed consistently with a temperature of 0.2 and is retried up to 3 times.

D Benchmark Details

HumanEval (Chen et al., 2021) is a foundational dataset for evaluating the functional correctness of

generated code. It consists of 164 hand-written programming problems, each including a function signature, a detailed docstring, and a set of hidden unit tests for evaluation. This benchmark is the standard for measuring the code generation capabilities of a model.

MBPP (Chen et al., 2022b) is a larger, crowd-sourced dataset. A critical challenge with the original MBPP dataset is that the prompts contain the ground-truth test cases, which can cause label leakage, particularly for baselines that perform selection or refinement. To ensure a fair comparison, we follow the setup of (Shinn et al., 2023). For our experiments, we use a representative subset of 175 problems sampled from the full dataset.

EvalPlus (Liu et al., 2023) ensures a rigorous and reliable evaluation by augmenting the standard test suites of both HumanEval and MBPP. HumanEval and MBPP sometimes pass solutions that are functionally correct on the provided tests but fail on more subtle edge cases. EvalPlus mitigates this risk by automatically generating a much larger and more comprehensive set of unit tests, providing a more robust measure of a candidate solution's true correctness.

xCodeEval (Khan et al., 2023) is more challenging, competition-level problems, which is sourced from the CodeForces platform. For our experiments, we sample a subset of 500 problems, after first filtering out any instances with incomplete or invalid test cases. A key feature of xCodeEval is its difficulty labels, which allow us to partition problems into four categories (Easy, Mid, Hard, and Expert) for a more fine-grained analysis of policy behavior.

Model	Input	Output
Qwen3_{4B}	0.11	0.42
Qwen3_{8B}	0.18	0.70
Qwen3_{14B}	0.35	1.40
Qwen3_{32B}	0.70	2.80
Llama3.1_{8B}	0.10	0.10
DeepSeek-Coder	0.14	0.28

Table 7: **Token pricing (USD per million tokens).** The values represent the cost for input and output, respectively.

E LLMs Details

We selected a range of state-of-the-art open-source language models to evaluate our PaT policy across different architectures and scales. All experiments were conducted using NVIDIA A 6000 GPUs, with models served via the vLLM framework at float16 precision. Due to its memory size, the Qwen3_{32B} model was run using 2-way tensor parallelism across two A6000 GPUs. Table 7 reports input and output token prices (USD per million tokens) collected from public provider listings[1]. Since there is no official price available for DeepSeek-Coder-V2-Lite-Instruct, we use the pricing policy of DeepSeek-Coder-V2 as a proxy.

Qwen3 (Yang et al., 2025) As our primary model family, we use the Qwen 3 series in four sizes: 4B, 8B, 14B, and 32B. For all experiments, we used the base, pre-trained versions of these models. Additionally, for all experiments with the Qwen3 model family, we prepended the /no_think command to all prompts.

Llama-3.1 (Dubey et al., 2024) To test the cross-family generalization of our approach, we use Llama-3.1-8B-Instruct. This is the instruction-tuned version of the Llama-3.1 8B model, optimized for following user commands and prompts.

DeepSeek-Coder-V2-Lite (Zhu et al., 2024) To evaluate on a model specifically fine-tuned for code generation, we use DeepSeek-Coder-V2-Lite-Instruct. This is the instruction-tuned "Lite" version of the DeepSeek-Coder-V2 family, with approximately 16B parameters.

F Baseline Details

Standard (Brown et al., 2020) represents the most direct approach to code generation, establishing the base capability of a model. It performs a single, one-time generation attempt to produce the entire program from the problem specification. For our code generation tasks, we follow a few-shot prompting protocol, providing the model with a small number of in-context examples to guide its output. We generate a single candidate solution with a low temperature of 0.3 to produce the most probable and deterministic result. This baseline serves to measure the model's raw performance

[1] https://artificialanalysis.ai (accessed 2025-09-25).

Method	Standard	Best-of-N	CodeT	FunCoder	PaT
Generation					
Samples (N)	1	5	11	1 + 10	5
Temperature	0.3	0.8	0.8	0.8	0.8
Retries	3	3	3	3	3
Planning					
Samples	-	-	-	1	1
Temperature	-	-	-	0.2	0.2
Retries	-	-	-	3	3
Test case generation & verification					
Benchmark-provided	X	O	X	O	O
Generation	X	X	O	O	O
Temperature	-	-	0.2	0.2	0.2
Retries	-	-	3	3	3

Table 8: **Baseline details.** Hyperparameter details for baselines and PaT.

without any complex strategies like sampling, refinement, or decomposition.

Best-of-N (Chen et al., 2021) aims to improve performance by exploring a diverse set of potential solutions through sampling. For each problem, it generates N candidate programs using a high temperature to encourage variety. To ensure a fair comparison with the trial phase of our PaT policy, we use the same sampling parameters: we set $N = 5$ and use a temperature of 0.8. Each of the 5 candidates is then executed against the provided test suite, and the one that passes the most tests is selected as the final solution. This method increases the probability of finding a correct solution at the cost of generating and evaluating multiple candidates. Since MBPP does not have a test suite, we deterministically submit the first of the 5 generated candidates.

CodeT (Chen et al., 2022a) is utilized from the implementation provided by the authors of FunCoder to ensure a faithful reproduction of their setup. The process begins by sampling a pool of candidate solutions. Following the hyperparameter settings of FunCoder, we used a larger sample size of $N = 11$ with a temperature of 0.8, which diverges from PaT and Best-of-N's $N = 5$. This is because CodeT's consensus-based ranking mechanism is highly dependent on a large and diverse candidate pool to function effectively; using a smaller sample size would artificially weaken this baseline.

FunCoder (Chen et al., 2024a) serves as our primary baseline for the Planning-before-Trial (PbT) policy, and we use the official implementation to ensure a fair and accurate comparison. The method operates as a rigid two-stage pipeline. First, a planner model decomposes the problem into a complete plan of helper functions using a temperature of 0.8. Only after this static plan is finalized does it proceed to the second stage, where it solves each subproblem using a consensus-based mechanism similar to CodeT, with $N = 10$ and a temperature of 0.8. This rigid, plan-first approach, where the plan is fixed regardless of generation outcomes, provides a direct contrast to our adaptive PaT policy.

G Prompt Details

Our prompting strategy largely follows the one provided in the FunCoder (Chen et al., 2024a) implementation to ensure a fair comparison. We use the same few-shot examples and sampling methods for both the generation (conquer) and planning (divide) phases.

G.1 Prompt for Generation

```
You are a programming copilot, you can solve a
    problem by writing Python functions. Your
    task is to:

  - For every turn, you need to write a Python
    function that returns the answer, based on
    current code (not code in chat history) and
    problem description.
  - Do not modify function name, arg names,
    docstring in given functions.
  - Consider reusing existing functions that are
     already implemented.
  - You can import libraries to better solve the
     problem.

<User>:

Current Code:
```

```
```python
def prime_factor(x: int) -> list:
    """get a list of prime factors of number $x$
    """
    ret = []
    i = 1
    while i * i <= x:
        i += 1
        if x % i == 0 and is_prime(i):
            ret.append(i)
    return ret

def is_prime(x: int) -> bool:
    """determine $x$ is a prime number or not"""
    if x < 2:
        return False
    for i in range(2, int(x**0.5) + 1):
        if x % i == 0:
            return False
    return True

def get_common(a: list, b: list) -> list:
    """get common element in two list $a$ and
    $b$"""
    ret = []
    for item in a:
        if item in b:
            ret.append(item)
    return ret

def sum_common_factors(a: int, b: int) -> int:
    """Return the sum of all common prime
    factors of $a$ and $b$"""

    raise NotImplementedError()
```

Let's think step by step and implement the following method `sum_common_factors` using existing functions to solve:
"Return the sum of all common prime factors of a and b"

<Assistant>:

First, I need to get the prime factors of a and b.
Second, I can use `for` loop to find common element in two factors list.
Finally, sum the common factor list and return the answer.
Here is the `sum_common_factors` function:

```
```python
def sum_common_factors(a: int, b: int) -> int:
    """Compute the sum of all common prime
    factors of $a$ and $b$"""
    factors_a = prime_factor(a)
    factors_b = prime_factor(b)
    common_factors = get_common(factors_a,
    factors_b)
    return sum(common_factors)
```

<User>:

Current Code:

```
```python
{prev_code}
```

Let's think step by step and implement the following method `{cur_func_name}` using existing functions to solve:
"{cur_func_doc}"

G.2 Prompt for Planning

```
You are a programming copilot, you can solve a
    problem by writing Python functions. Your
    task is to:
  - The previous attempt to direct implement the
     target function is failed, indicating its
    overall logic might be too complex to
    implement directly.
  - For every turn, you need to write a Python
    function that returns the answer based on
    Current Code (not code in chat history).
  - Do not modify function name, arg names,
    docstring in given functions.
  - You can import libraries to better solve the
     problem.
  - You can leave new function unimplemented for
     now, but write the function at the end of
    the code and comment what the function does.
  - Therefore, you must decompose it into
    multiple smaller, manageable helper
    functions.

<User>:

Current Code:
```python
def sum_common_factors(a: int, b: int) -> int:
    """Compute the sum of all common prime
    factors of $a$ and $b$"""
    raise NotImplementedError()
```

Let's think step by step and complete the
    following Python function `
    sum_common_factors` that solves:
"Compute the sum of all common prime factors of
    a and b"

<Assistant>:

First, I need to get the prime factors of a
    and b.
Second, I can use `for` loop to find common
    element in two factors list.
Finally, sum the common factor list and return
    the answer.
Here is the `sum_common_factors` function:

```python
def sum_common_factors(a: int, b: int) -> int:
    """Compute the sum of all common prime
    factors of $a$ and $b$"""
    factors_a = prime_factor(a)
    factors_b = prime_factor(b)
    common_factors = get_common(factors_a,
    factors_b)
    return sum(common_factors)

def prime_factor(x: int) -> list:
    """get a list of prime factors of number $x$
    """
    raise NotImplementedError()
```
```
```

```
def get_common(a: list, b: list) -> list:
    """get common element in two list $a$ and
    $b$"""
    raise NotImplementedError()
'''

<User>:

Current Code:
```python
def sum_common_factors(a: int, b: int) -> int:
    """Compute the sum of all common prime
    factors of $a$ and $b$"""
    factors_a = prime_factor(a)
    factors_b = prime_factor(b)
    common_factors = get_common(factors_a,
    factors_b)
    return sum(common_factors)

def get_common(a: list, b: list) -> list:
    """get common element in two list $a$ and
    $b$"""
    raise NotImplementedError()
'''

Let's think step by step and complete the
    following Python function `get_common` that
    solves:
"get common element in two list $a$ and $b$"

<Assistant>:

Here is the `get_common` function:

```python
def get_common(a: list, b: list) -> list:
    """get common element in two list $a$ and
    $b$"""
    ret = []
    for item in a:
        if item in b:
            ret.append(item)
    return ret
'''

<User>:

Current Code:
```python
{prev_code}
'''

Let's think step by step and complete the
    following Python function `{cur_func_name}`
    that solves:
"{cur_func_doc}"
```
```