How Requirement Formats Affect Large Language Models for Code Generation: Empirical Study

Anonymous authors
Paper under double-blind review

Abstract

Large language models (LLMs) generate code from requirements expressed as prose, scenarios, tests, or specifications, yet the effect of these formats remains unclear. We compare 13 requirement formats across four programming benchmarks and five LLMs, then manually verify transformed requirements to examine whether rewriting errors explain the observed differences. Across model–benchmark pairs, the highest and lowest Pass@1 scores under automatic transformation differ by 6.34%–32.72%. Manual correction raises mean Pass@1 by 1.44%, but substantial differences among formats remain: semi-structured formats lead on average, while preferred formats vary by model and algorithmic task. We use these findings to develop ReForm, which estimates model-specific format preferences from execution outcomes grouped by algorithmic subcategory and selects a format for each new requirement. On held-out OJBench, ReForm achieves 30.00% mean Pass@1 across five models, compared with 22.84% for original requirements and 28.10% for the best single format measured on OJBench. For Claude-Sonnet-4.6, the selected formats also improve Pass@1 when combined with Self-Planning, SCoT, and OpenCode. These results show that requirement-format choice matters beyond transformation fidelity and that format selection transfers to new problems and downstream coding workflows. Our source code and details are publicly available at https://figshare.com/s/b8c47472426b87eb7776.

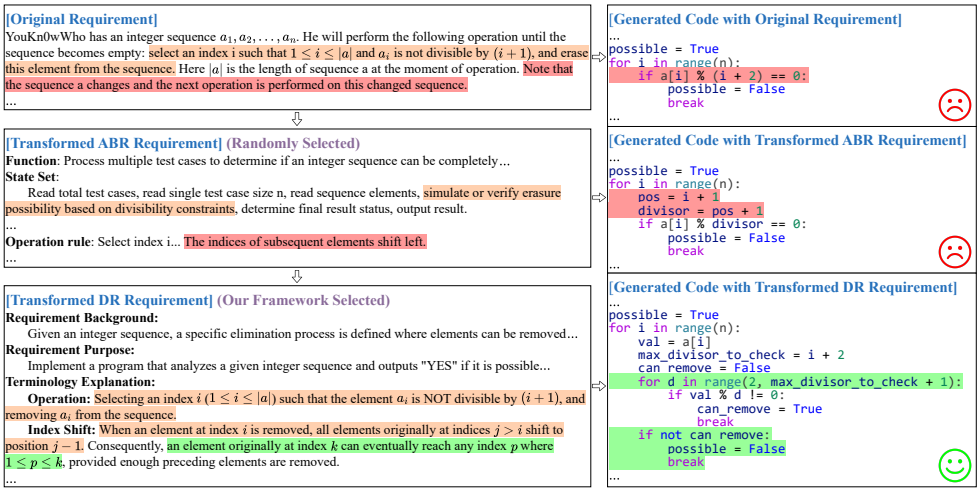


Figure 1: Generated code based on different requirement formats for a sequence-erasure problem from CodeContests benchmark with Qwen3.5-Plus. The original requirement and the Automaton-Based Requirement (ABR) formats yield programs that check only the initial element positions. The Dimensional Requirement (DR) format yields a program that checks divisors associated with possible shifted positions and passes the evaluation tests.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

Translating requirements into executable programs has become an important application of large language models (LLMs) (Jain et al., 2025; Jang et al., 2026). In the 2025 Stack Overflow Developer Survey, 50.6% of surveyed professional developers report using AI tools daily in their development workflows (Stack Overflow, 2025). Requirements are the interface between a programming task and the model asked to solve it. The same task may be presented as prose, a behavior-driven scenario (Smart, 2014), or a formal specification (Wing, 1990a), with different ways of organizing its inputs, constraints, and expected behavior. This variety raises two research questions. **Q1: Does requirement format affect code-generation correctness beyond errors introduced by rewriting?** If so, we examine whether preferred formats change with the target model and the requirement's algorithmic meaning. **Q2: Can these preferences guide format selection for held-out problems?**

Prior work establishes that the wording and presentation of a task can matter, but does not resolve this sequence of questions for software requirement formats. **Requirement reformulation.** Sarker et al. (2026) and Wang et al. (2023) examine localized syntactic changes to code-generation prompts. StoryCoder (Jang et al., 2026) turns programming problems into narratives, while PEARL (Fu et al., 2024) studies paraphrasing for question answering. Studies of prompt variation also report that semantically similar inputs can produce different code and that models may prefer different styles (Paleyes et al., 2025; Midolo et al., 2026). However, these studies mainly test robustness to small changes or propose a single new presentation. They do not compare many requirement formats under the same setting, nor ask which format suits a given model and problem. **Reasoning strategies and coding agents.** CoT (Wei et al., 2022), SCoT (Li et al., 2023), and Self-Planning (Jiang et al., 2024) structure the steps from a requirement to code. Coding agents such as OpenHands (Wang et al., 2025a), A Self-Improving Coding Agent (Robeyns et al., 2025), and USEagent (Applis et al., 2025) extend this process with iterative or tool-based workflows. These methods change how the model reasons or acts after reading the requirement, but keep the requirement itself unchanged. Since format selection works at an earlier step, we therefore also test whether format selection complements reasoning strategies and coding agents.

To answer Q1, we express each programming problem in 13 requirement formats, including its original requirement, organized into unstructured, semi-structured, and structured families. Across five models and four benchmarks, we hold the code-generation instruction and evaluation tests fixed while comparing Pass@1. Under automatic transformation, the spread between formats ranges from 6.34% to 32.72% across model–benchmark pairs. Because reformulation can alter a problem's meaning, we manually check and correct the transformed requirements before repeating the evaluation. Errors affect 5.73%–18.53% of transformations across models, and correction increases mean Pass@1 by 1.44%; substantial differences among formats remain. Rewriting errors therefore do not account for the observed variation alone. The preferred format also changes with the target model and the requirement's algorithmic category, motivating a choice tailored to both. Figure 1 shows an example of how requirement formats affect LLMs for code generation.

To answer Q2, we develop REFORM, a requirement-format selection method that uses these model- and task-dependent preferences. We classify calibration problems into 31 terminal algorithmic labels and aggregate code execution outcomes for every format within each label and target model. For a new requirement, REFORM predicts its label, retrieves the format with the highest observed pass rate for that model, and applies its transformation template when needed. The target model then produces one program from the selected format. The recommendation uses prior problems' outcomes, not test results from the new problem. We test transfer on OJBench problems excluded from the recommendation knowledge base. Across five models, REFORM reaches 30.00% mean Pass@1, compared with 22.84% for original requirements and 28.10% for the best single format measured on OJBench. We also pair the selected format with Self-Planning, SCoT, and the OpenCode agent on OJBench using Claude-Sonnet-4.6. Relative to each workflow using original requirements, Pass@1 increases by 8.30%, 6.03%, and 3.02%, respectively. Thus, format selection remains useful alongside reasoning strategies and an agent. These evaluations concern self-contained algorithmic problems, not repository-level changes.

Our contributions are as follows:

- **Controlled format comparison.** We compare 13 formats across five LLMs and four benchmarks. Under automatic transformation, within-pair Pass@1 spreads span 6.34%–32.72%; after correcting transformation errors, substantial format differences remain.

- **Model- and task-aware selection.** REFORM learns format preferences from execution outcomes by target model and algorithmic subcategory, then selects and prepares a format for each new requirement.
- **Transfer and workflow evaluation.** On held-out OJBench (Wang et al., 2025b), REFORM raises mean Pass@1 from 22.84% (original requirements) to 30.00%. Its selected formats also benefit Claude-Sonnet-4.6 when paired with Self-Planning, SCoT, and OpenCode.

2 BACKGROUND

2.1 REQUIREMENT FORMATS

Software requirement formats can primarily fall into three categories: unstructured, semi-structured, and structured formats. For example, semi-structured formats predominate in modern practice; in agile development (which accounts for over 80% of software workflows) more than 85% of teams rely on User Stories for daily work (Lucassen et al., 2016; Méndez Fernández et al., 2017). Similarly, Dimensional Requirement (DR) based on ISO/IEC/IEEE 29148 (ISO/IEC/IEEE, 2018) and Test-Driven Development (TDD) specifications are widely used in DevSecOps. Meanwhile, structured formats, such as requirement contracts and formal specifications, provide unambiguous semantics for synthesis and verification in high-reliability systems. We analyze 13 mainstream requirement formats from 3 types (unstructured, semi-structured, and structured formats). Details for each format are in Appendix B. Some requirement formats were originally developed for system-level requirements engineering. We adapt them to function-level programming tasks by mapping users, goals, and hierarchical abstractions to computational entities, objectives, and program components.

Unstructured Formats. Unstructured formats represent requirements using free-form natural language descriptions without predefined schematic layout constraints, reflecting how users informally articulate tasks. **Original Requirement (OR)** retains the natural-language statement from standard coding benchmarks. **Narrative Requirement (NR)** embeds the user intent into a story-driven narrative, mapping entities and rules to story objects introduced by Jang et al. (2026). **Pain-Point Description (PPD)** describes users' operational difficulties, their consequences, desired system capabilities, and acceptance outcomes. PPD is a study-specific template motivated by the progression from user needs and difficulties to stakeholder requirements (Hull et al., 2011, Section 2.2).

Semi-Structured Formats. Semi-structured formats organize natural-language descriptions into predefined template fields or conceptual hierarchies to improve clarity and component separation. **User Story (US)** specifies a user role, desired capability, and business value, supplemented by explicit scenarios and acceptance criteria (Cohn, 2004). **Dimensional Requirements (DR)** explicitly disaggregates the requirement into different elements, including background, purpose, terminology, inputs, outputs, examples, edge cases, and constraints. These dimensions build on established foundations in requirements engineering (Glinz, 2000; Greenspan et al., 1994) and align with the IEEE recommended practice (Doe, 2011). **Scenario-Driven Requirement (SDR)** describes the context of use and the expected sequence of interactions, drawing on scenario-based requirements engineering (Sutcliffe, 2002). Our template makes the context, participating actors, triggering events, and expected behavior explicit. **Layered Requirement (LR)** organizes requirements at multiple levels of abstraction, following hierarchical requirements development (Hull et al., 2011, Chapter 2). Our template links the task's needs and objectives to product, feature, function, and component levels. **Goal-Driven Requirement (GDR)** expresses how the system goals are decomposed into structured sub-goals through AND/OR refinement relations (van Lamsweerde, 2001). **Behavior-Driven Requirement (BDR)** expresses functionality using the standard Behavior-Driven Development (BDD) specification with feature, background, and Given-When-Then scenario blocks (Smart, 2014). **Test-Driven Requirement (TDR)** emphasizes functional specifications through concrete input-output test cases, boundary assertions, and expected dynamic behaviors (Beck, 2002).

Structured Formats. Structured formats encode requirements using formal state machines, mathematical logic, or program contracts to eliminate semantic ambiguity. **Automaton-Based Requirement (ABR)** explicitly formalizes system behaviors via finite states, inputs, outputs, and explicit state transition rules (Hopcroft et al., 2001). **Contract-Based Requirement (CBR)** specifies function semantics using formal Design-by-Contract primitives, including preconditions, postconditions, and class invariants (Meyer, 1992). **Formal Specification (FS)** presents requirements using formal program specification syntax (*e.g.,* Dafny-style constructs like `requires`, `ensures`, and

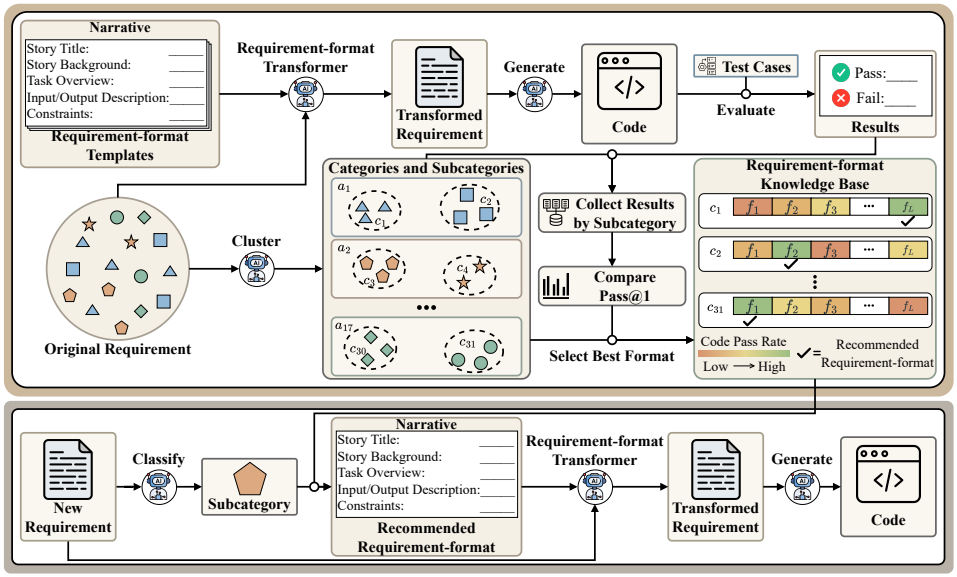


Figure 2: Overview of REFORM. Top: evaluate the effect of requirement formats and aggregate them to construct a recommendation knowledge base. Bottom: given an input problem, recommend a suitable format, reformulate the requirement, and generate code.

`modifies`) (Wing, 1990b; Leino, 2010). FS denotes a prompt representation format and does not imply static formal verification.

2.2 Related Work

Recent work on LLM-based code generation improves solution quality through planning and reasoning (Jiang et al., 2024; Li et al., 2023; Wei et al., 2022), modular self-revision (Le et al., 2024), and multi-agent workflows (Hong et al., 2024; Wang et al., 2025a). Despite these advances, they primarily aim to enhance LLMs' performance based on random requirement formats, overlooking how requirement formats affect LLMs in user intent comprehension and code synthesis. Recent work begins to explore localized syntactic perturbations (Sarker et al., 2026; Wang et al., 2023), prompt translation (Fu et al., 2024), and narrative-driven problem reformulations (Jang et al., 2026). These studies confirm that semantically equivalent prompts induce substantial code generation variance and distinct model preferences (Midolo et al., 2026; Paleyes et al., 2025). However, they focus on ad-hoc prompt edits rather than systematically investigating software requirement formats. Different from these studies, we systematically study the effect of requirement formats on LLMs in programming and design a framework to select suitable formats for LLMs.

3 Effect of Requirement Formats on LLMs for Code Generation

3.1 Implementations

To systematically investigate the impact of requirement formats on LLM-based code generation, we conduct experiments on diverse benchmarks and advanced LLMs with standardized metrics.

Benchmarks and Models. We compare formats on four algorithmic code-generation benchmarks: APPS (300 problems) (Hendrycks et al., 2021), CodeContests (165) (Li et al., 2022), xCodeEval (300) (Khan et al., 2024), and LiveCodeBench-Lite (175) (Jain et al., 2025). For transfer evaluation, we use 232 OJBench problems (Wang et al., 2025b), excluded from recommendation construction. All tasks are self-contained and evaluated through executable programs; they do not cover repository-level software changes. The open-weight models are DeepSeek-V4-Pro (DeepSeek-AI, 2026) and GLM-5 (Z.AI, 2026). The closed-weight models are Qwen3.5-Plus (Alibaba Cloud, 2026), Claude-Sonnet-4.6 (Anthropic, 2026), and Gemini-3-Flash (Doshi & Gemini Team, 2025).

Experimental Details. We set the sampling temperature to 0.0 in all experiments, with a maximum context output length of $8,192$ tokens. All prompts for requirement transformations and code generation are provided in Appendix A.

Table 1: Performance of LLMs across the 13 requirement formats in Pass@1 (%). Within each model and benchmark, the highest result is highlighted in indigo and the lowest result in amber.

LLM	Benchmark	Requirement Format (%)													Δ (%)
		Unstructured			Semi-structured							Structured			
		OR	NR	PPD	US	DR	SDR	LR	GDR	BDR	TDR	ABR	CBR	FS	
DeepSeek-V4-Pro	CodeContests	41.21	36.97	43.67	30.30	**48.48**	44.85	42.42	36.36	**24.85**	40.00	44.24	38.18	41.21	**23.63**
	LiveCodeBench	48.57	46.86	44.57	**42.29**	54.86	56.00	**59.43**	58.86	55.43	54.29	54.86	54.86	53.14	**17.14**
	APPS	63.67	66.33	**58.33**	62.00	66.33	62.33	64.00	64.33	61.00	61.00	**68.67**	66.67	64.33	**10.34**
	xCodeEval	63.67	62.67	53.67	**52.67**	**64.33**	58.00	58.00	60.00	53.33	57.67	60.00	60.00	58.67	**11.66**
Qwen3.5-Plus	CodeContests	43.03	44.24	41.21	43.03	**54.55**	50.91	46.06	48.48	32.12	41.21	39.39	41.21	**30.30**	**24.25**
	LiveCodeBench	46.29	**42.29**	42.86	44.00	**63.43**	57.14	61.14	60.00	57.14	57.14	58.29	58.86	49.14	**21.14**
	APPS	38.33	38.33	53.67	**37.67**	**55.33**	53.00	54.67	54.67	52.67	54.33	53.33	53.00	49.00	**17.66**
	xCodeEval	40.67	**38.33**	51.33	38.67	50.33	**52.33**	49.67	50.67	50.33	48.67	47.33	49.33	49.00	**14.00**
GLM-5	CodeContests	28.48	26.67	20.00	**15.15**	**32.73**	30.91	27.88	25.45	18.79	26.06	27.27	25.45	23.03	**17.58**
	LiveCodeBench	48.00	43.43	42.86	**40.57**	48.57	49.71	47.43	**50.86**	45.71	45.71	46.86	**50.86**	45.14	**10.29**
	APPS	62.67	57.33	55.67	**50.67**	**63.67**	61.00	59.33	58.33	56.33	61.33	61.67	58.00	56.00	**13.00**
	xCodeEval	51.00	48.67	45.67	**43.33**	49.67	48.67	51.33	50.33	**52.33**	**52.33**	48.67	50.33	45.00	**9.00**
Claude-Sonnet-4.6	CodeContests	29.09	26.06	21.21	26.67	**35.15**	26.67	28.48	27.88	24.85	33.33	29.09	30.30	**20.61**	**14.54**
	LiveCodeBench	**39.43**	48.57	40.57	43.43	50.86	47.43	**53.14**	52.00	40.00	41.14	**39.43**	42.86	40.00	**13.71**
	APPS	65.67	63.33	58.00	58.67	59.33	58.33	58.33	**55.67**	61.00	64.00	**68.00**	65.67	59.33	**12.33**
	xCodeEval	60.33	60.67	56.67	**54.33**	64.33	60.67	62.33	63.67	60.33	63.00	62.67	**65.00**	57.00	**10.67**
Gemini-3-Flash	CodeContests	58.79	39.39	36.36	**29.70**	60.61	52.73	51.52	44.85	43.03	**62.42**	40.61	32.12	37.58	**32.72**
	LiveCodeBench	56.00	60.00	**50.86**	55.43	73.14	69.14	70.86	67.43	66.85	**75.43**	72.57	70.85	64.00	**24.57**
	APPS	**83.33**	81.33	79.33	78.33	82.00	83.00	82.00	**75.67**	81.33	81.33	81.00	82.67	78.00	**7.66**
	xCodeEval	77.33	77.00	75.67	76.00	**78.67**	76.33	77.67	**72.33**	74.67	77.33	77.67	76.67	75.00	**6.34**
Average (%)		52.28	50.42	48.61	**46.15**	**57.82**	54.96	55.28	53.89	50.60	54.89	54.08	53.64	49.77	**11.67**

Metrics. Following previous works Li et al. (2023); Jiang et al. (2024), we report Pass@1 (%) as the primary metric, measured as the percentage of problems solved correctly with one generated solution per problem. A solution is considered correct only if it passes all evaluation tests without runtime errors or timeouts.

3.2 Requirement Format Transformation and Evaluation

The same programming task can be expressed in different requirement formats: a narrative connects its objects and rules through a story, whereas software requirement elements separate its inputs, outputs, and constraints into explicit fields. Starting from each original requirement, we express it in the formats of Section 2.1 while preserving the requirement task semantics across all formats. To examine how these formats affect code generation, we prepare each requirement in all 13 formats, then generate and execute code from each format. The manual verification, described in Section 3.3, checks whether the transformed requirements remain consistent with the original requirement.

Transforming requirements. Let x_i denote the original requirement of problem i, m the target LLM, and $\mathcal{F} = \{f_1, \ldots, f_{13}\}$ the set of templates corresponding to the 13 requirement formats. Let p_{INS} be the shared transformation instruction, and let $\widetilde{x}_{i,m,f}$ denote the requirement in the format specified by $f \in \mathcal{F}$ supplied to code generation. For a transformed format, the model receives the instruction, the selected template, and the original requirement:

$$\widetilde{x}_{i,m,f} = m(p_{\text{INS}}, f, x_i). \tag{1}$$

A requirement represented by the original requirement is used directly. The shared instruction requires preservation of the task objective and behavior, input/output specifications, constraints, examples, and edge cases, while prohibiting solutions, code, and unsupported assumptions. The shared instruction and individual templates are provided in Appendices A and B.

Generating and evaluating code. The same model m receives $\widetilde{x}_{i,m,f}$ in a separate call with the shared code-generation instruction p_{code}, which requests a solution. We extract the program from the response and evaluate it using the tests associated with the original problem:

$$g_{i,m,f} = m(p_{\text{code}}, \widetilde{x}_{i,m,f}), \qquad z_{i,m,f} = E_i(g_{i,m,f}) \in \{0, 1\}. \tag{2}$$

Here, $g_{i,m,f}$ denotes the generated program after removing any surrounding code fences, and E_i is the execution-based evaluator, not an LLM. It returns 1 only when the program passes all evaluation tests; an incorrect output, execution error, or timeout yields 0. The tests remain fixed across formats and are not used as feedback to revise requirements or generated programs. Each problem–model–format combination receives one code-generation attempt under the settings in Section 3.1.

Table 2: Pass@1 (%) for the 13 requirement formats before and after manual semantic verification of automatically transformed requirements. Unvalidated and Validated are row-wise means across all 13 formats before and after verification. Format definitions follow Section 2.1.

LLM	Benchmark	Requirement Format (%)													Δ (%)	Average (%)	
		Unstructured			Semi-structured							Structured				Unvalidated	Validated
		OR	NR	PPD	US	DR	SDR	LR	GDR	BDR	TDR	ABR	CBR	FS			
DeepSeek-V4-Pro	CodeContests	41.21	41.82	44.85	36.97	**50.30**	45.45	43.03	36.97	**26.06**	40.61	46.67	40.00	41.82	**24.24**	39.44	**41.21**
	LiveCodeBench	48.57	48.57	46.86	**45.71**	55.43	58.29	60.00	**60.57**	57.14	56.00	**60.57**	55.43	54.29	**14.86**	52.62	**54.42**
	APPS	63.67	68.67	67.00	68.67	**73.67**	64.33	65.00	65.67	65.33	**62.00**	70.67	68.00	65.33	**11.67**	63.77	**66.77**
	xCodeEval	63.67	64.33	55.33	58.33	**68.33**	60.33	59.00	60.67	**54.00**	58.00	60.00	60.33	61.00	**14.33**	58.67	**60.26**
Qwen3.5-Plus	CodeContests	43.03	45.45	42.42	43.64	**54.55**	50.91	46.67	49.09	33.33	41.82	39.39	41.21	**31.52**	**23.03**	42.75	**43.31**
	LiveCodeBench	46.29	**44.57**	45.14	45.71	**69.14**	58.29	61.71	61.71	60.00	57.14	59.43	60.57	50.86	**24.57**	53.67	**55.43**
	APPS	38.33	38.33	53.33	**37.67**	**78.67**	53.00	54.67	54.33	52.67	54.33	53.33	53.00	49.00	**41.00**	49.85	**51.59**
	xCodeEval	40.67	**38.33**	51.33	38.67	**74.00**	52.33	49.67	50.67	50.33	48.67	47.33	49.33	49.00	**35.67**	47.44	**49.26**
GLM-5	CodeContests	28.48	27.27	22.42	21.21	**35.15**	32.73	27.88	26.67	**19.39**	26.06	27.88	27.27	23.64	**15.76**	25.22	**26.62**
	LiveCodeBench	48.00	**43.43**	**43.43**	**43.43**	**52.00**	50.86	48.57	**52.00**	45.71	46.86	48.57	50.86	48.57	**8.57**	46.59	**47.87**
	APPS	62.67	58.00	56.67	**56.00**	**68.67**	62.33	60.33	59.33	58.00	61.33	62.00	58.33	58.67	**12.67**	58.62	**60.18**
	xCodeEval	51.00	49.33	**47.67**	**47.67**	**57.00**	50.67	54.33	51.67	54.00	52.67	50.33	51.33	49.00	**9.33**	49.03	**51.28**
Claude-Sonnet-4.6	CodeContests	29.09	27.27	21.82	26.67	**36.36**	27.88	31.52	28.48	24.85	34.55	29.70	30.30	**21.21**	**15.15**	27.65	**28.44**
	LiveCodeBench	**39.43**	49.14	46.86	45.14	50.86	48.57	**53.71**	52.00	41.71	42.29	40.00	44.57	42.86	**14.28**	44.53	**45.93**
	APPS	65.67	64.33	59.33	59.33	62.00	59.00	59.67	**58.33**	61.67	65.33	**70.00**	67.67	60.67	**11.67**	61.18	**62.54**
	xCodeEval	60.33	62.00	58.00	**56.33**	64.67	62.00	62.67	64.67	61.67	63.67	63.00	**65.67**	58.00	**9.34**	60.85	**61.74**
Gemini-3-Flash	CodeContests	58.79	39.39	36.97	**30.30**	61.21	52.73	52.73	45.45	44.85	**63.03**	41.21	32.12	38.79	**32.73**	45.36	**45.97**
	LiveCodeBench	56.00	61.14	**53.71**	57.71	73.14	69.14	72.57	68.57	68.00	**75.43**	74.29	72.57	67.43	**21.72**	65.58	**66.90**
	APPS	83.33	83.00	80.00	79.33	83.33	**83.67**	82.67	**79.00**	82.67	83.00	82.33	**83.67**	80.00	**4.67**	80.72	**82.00**
	xCodeEval	77.33	78.00	76.33	76.67	78.67	76.67	**79.00**	**73.33**	75.00	77.33	77.67	78.00	76.33	**5.67**	76.33	**76.95**
Average (%)		52.28	51.62	50.47	**48.76**	**62.36**	55.96	56.27	54.96	51.82	55.51	55.22	54.51	51.40	**13.60**	52.49	**53.93**

Keeping the model, generation prompt, and tests fixed within each comparison allows us to compare alternative requirement formats under the same generation and evaluation procedure.

Requirement format is a consequential choice. Across model–benchmark pairs, the spread between the strongest and weakest format ranges from 6.34% to 32.72%. The contrast is especially clear for Gemini-3-Flash on CodeContests, where Pass@1 increases from 29.70% with US to 62.42% with TDR, whereas on xCodeEval the same model varies only from 72.33% with GDR to 78.67% with DR. Thus, format sensitivity is substantial. The full comparison is given in Table 1.

Among the three format families, semi-structured formats have the highest average (53.37%), followed by structured (52.50%) and unstructured formats (50.44%). This aggregate ordering masks substantial variation within a family: DR has the highest individual-format average with 57.82% Pass@1, whereas US only achieves 46.15%, the lowest performance in nine of the 20 pairs. In the structured family, FS averages 49.77%, below NR (50.42%) and DR (57.82%). The results thus point to information organization, not the amount of structure alone, as an important factor in code generation. Rankings also depend on the model and benchmark: DR ranks first for Qwen3.5-Plus on three benchmarks, TDR for Gemini-3-Flash on two, and no format leads across all 20 pairs.

3.3 Manual Verification of Transformed Requirements

Automatic transformation might introduce semantic or formatting deviations even when the transformation instruction requires fidelity to the original requirement. We therefore manually inspect each transformed requirement against its original requirement and correct any discrepancy in the task objective, expected behavior, inputs, outputs, constraints, examples, edge cases, or template organization. We then repeat the evaluation in Section 3.2 with the verified requirements, keeping the target model, code-generation prompt, and tests fixed. Table 2 compares the scores before and after verification for the same 20 model–benchmark pairs with 13 requirement formats.

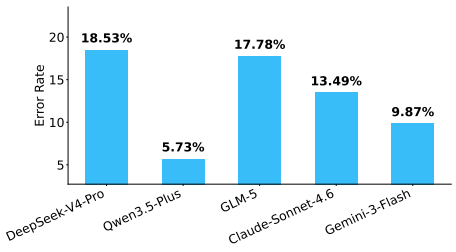


Figure 3: Error rate of automatically transformed requirements by LLM.

When the manually verified requirements are used as inputs, the average Pass@1 increases from 52.49% to 53.93%, an increase of 1.44%. For example, DeepSeek-V4-Pro on APPS increases from 63.77% to 66.77%, while Qwen3.5-Plus on CodeContests increases from 42.75% to 43.31%. The row mean increases in every pair, although individual format scores do not all improve.

Despite this improvement, the overall gain is modest. Manual verification identifies errors in 5.73%–18.53% of automatically transformed requirements across the five LLMs (Figure 3). Together, these results suggest that the automatically generated requirement formats are generally reliable, possibly because current LLMs are already capable of preserving most task information during reformulation. Format sensitivity nevertheless remains pronounced after verification: on APPS, Qwen3.5-Plus reaches 78.67% with DR but only 37.67% with US, a spread of 41.00%.

The family-level ordering is unchanged after verification: semi-structured formats have the highest average (55.09%), followed by structured (53.71%) and unstructured formats (51.46%). DR has the highest individual-format average (62.36%). Across the 20 model–benchmark pairs, DR attains the highest Pass@1 among the 13 formats in 12 pairs. US has the lowest individual-format average (48.76%) and ranks last in seven pairs. DR is not universally best: it leads for Qwen3.5-Plus on all four benchmarks, while TDR leads for Gemini-3-Flash on CodeContests and LiveCodeBench-Lite. The persistence of these differences before and after verification suggests that transformation errors alone do not account for the observed variation in code-generation performance.

> **Answer to Q1.** Requirement format affects code-generation performance, and its effect varies across models and benchmarks. Manual semantic verification improves the average score by 1.44%, while substantial differences among formats remain. Thus, both transformation fidelity and format choice matter when presenting requirements to code-generation models.

4 REFORM METHODOLOGY

Our analysis shows that requirement format affects code-generation performance and that a useful format can depend on what the problem asks for. Given a requirement, REFORM therefore identifies its algorithmic meaning and recommends a suitable way to express it for code generation. Figure 2 shows the two stages: constructing a recommendation knowledge base from observed execution outcomes, and using that knowledge base to reformulate a new requirement before code generation.

4.1 REQUIREMENT FORMAT RECOMMENDATION

The recommendation begins with the algorithmic content of a requirement. We classify each pooled requirement by its core task, using the objects, operations, state dependencies, and objectives stated in the problem. A classifier LLM assigns it to one of 17 top-level categories and then to one of 31 terminal classification labels in the taxonomy $\mathcal{K}$. Appendix C lists all 17 categories and 31 labels. For each target model, the recommendation knowledge base uses the same pool of problems from the four benchmarks in Section 3.1 and stores execution outcomes by classification label. We write $m_{\text{cls}}(p_{\text{cls}}, x, \mathcal{K})$ for the label assigned by the classifier model given its classification instruction, the requirement, and the taxonomy. The subscript distinguishes this classification role from the target model m used for transformation and code generation.

Let $\mathcal{D}_{\text{cal}}$ denote the problem set. For each subcategory c, we collect its problem indices as

$$I_c = \{i : x_i \in \mathcal{D}_{\text{cal}},\ m_{\text{cls}}(p_{\text{cls}}, x_i, \mathcal{K}) = c\}. \tag{3}$$

Using the execution outcomes $z_{i,m,f}$ defined in Equation 2, we estimate the pass rate of each format within every nonempty subcategory and select a format with the highest observed score:

$$\widehat{\mu}(m, c, f) = \frac{1}{|I_c|} \sum_{i \in I_c} z_{i,m,f}, \qquad \rho(m, c) \in \arg\max_{f \in \mathcal{F}} \widehat{\mu}(m, c, f). \tag{4}$$

Here, $\widehat{\mu}(m, c, f)$ measures how often model m solves a problem in subcategory c when given format f, and $\rho(m, c)$ stores the corresponding format recommendation. Each problem in I_c contributes equally to this estimate.

4.2 CODE GENERATION WITH RECOMMENDED REQUIREMENT FORMAT

Given a new requirement x and target model m, REFORM first assigns the requirement to a subcategory and retrieves the stored recommendation:

$$c = m_{\text{cls}}(p_{\text{cls}}, x, \mathcal{K}), \qquad f^*(x, m) = \rho(m, c). \tag{5}$$

Following the procedure in Section 3.2, we generate the recommended requirement format $\widetilde{x}$ with the selected format $f^*(x, m)$, applying Equation 1 when transformation is needed. The target model

Table 3: REFORM results (Pass@1) on OJBench. OR denotes the original requirement. Best fixed is the highest-scoring single format for each model; REFORM selects a format per problem.

Benchmark	Model	OR	Best fixed	REFORM
OJBench	Claude-Sonnet-4.6	14.22	21.12	23.28 ($\Delta_{\text{OR}} = +9.06$; $\Delta_{\text{fixed}} = +2.16$)
	DeepSeek-V4-Pro	22.84	31.47	34.48 ($\Delta_{\text{OR}} = +11.64$; $\Delta_{\text{fixed}} = +3.01$)
	Gemini-3-Flash	25.43	33.62	36.21 ($\Delta_{\text{OR}} = +10.78$; $\Delta_{\text{fixed}} = +2.59$)
	GLM-5	21.55	23.28	25.43 ($\Delta_{\text{OR}} = +3.88$; $\Delta_{\text{fixed}} = +2.15$)
	Qwen3.5-Plus	30.17	31.03	30.60 ($\Delta_{\text{OR}} = +0.43$; $\Delta_{\text{fixed}} = -0.43$)
Average (%)		22.84	28.10	**30.00** ($\Delta_{\text{OR}} = +7.16$; $\Delta_{\text{fixed}} = +1.90$)

then generates code with the code-generation instruction p_{code}:

$$\widehat{g} = m(p_{\text{code}}, \widetilde{x}). \tag{6}$$

We evaluate the transfer of these recommendations on OJBench (Wang et al., 2025b). Its problems are excluded from knowledge-base construction. The fixed taxonomy and model-specific recommendations are applied to each OJBench requirement for format selection.

Table 3 reports the OJBench results. REFORM reaches 30.00% mean Pass@1 across the five evaluated models, compared with 22.84% for OR and 28.10% for the best fixed format selected per model on OJBench, differences of 7.16% and 1.90%, respectively. It exceeds the best fixed format in four of the five model rows, with a 0.43% decrease in the remaining row. These results support the transfer of meaning-guided format selection to independent competitive-programming problems, while the gain is not uniform. The end-to-end evaluation does not independently verify whether each predicted algorithmic subcategory is correct, but the observed Pass@1 gains support the effectiveness of the overall automated pipeline.

All candidate formats follow the same recommendation and code generation procedure. During transformation, the selected template determines the requirement's organization, while the shared instruction specifies the preservation requirements described earlier. Format selection can thus vary across problems, requiring only one selected format and one code-generation attempt each.

Answer to Q2. Selecting requirement formats according to the algorithmic meaning of each problem yields 30.00% mean Pass@1 on held-out OJBench, compared with 22.84% for original requirements and 28.10% for the best fixed format. This supports adapting the requirement format to the task type within the standalone programming setting evaluated here.

5 EMPIRICAL LESSONS

Figure 4 reveals how preferred requirement formats vary across models and problem categories. We derive three practical lessons for expressing requirements for code generation.

Express task-critical concepts clearly. DR is recommended across many categories, including 29 of 31 subcategories for Qwen3.5-Plus. Its broad applicability may stem from its distinct fields: terminology can clarify task entities, input/output fields can specify interfaces, and constraints and edge cases can expose conditions essential to correctness. Different fields may support different problem types, although their individual contributions are not isolated. When writing requirements, make the concepts and conditions central to the task explicit, using dedicated fields where helpful.

Choose useful requirement formats rather than maximal structure. Even within one model, preferred formats span different levels of structure. For Claude-Sonnet-4.6, Figure 4 recommends NR for category 6.1, TDR for 5.3, and CBR for 3.1, covering unstructured, semi-structured, and structured representations. Thus, increasing formality does not consistently produce the preferred representation. Consider narrative explanations, concrete examples, or explicit contracts according to the problem, rather than expressing every requirement in the most formal available format.

Match the format to both the model and requirement meaning. Similar overall code-generation performance does not imply similar format preferences. Claude-Sonnet-4.6 and GLM-5 have comparable mean Pass@1 with original requirements across the four benchmarks (48.63% and

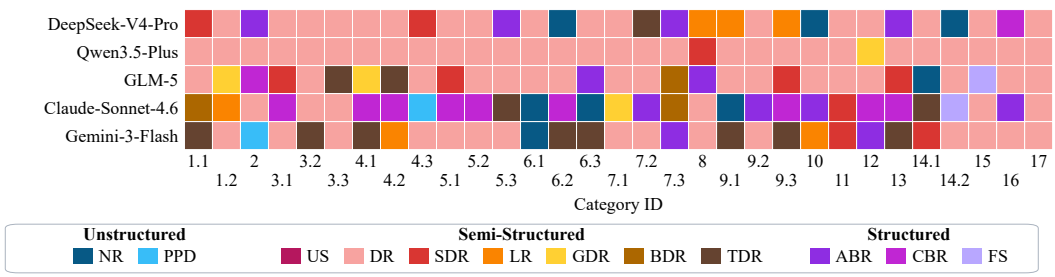


Figure 4: Best-performing single requirement formats across models and algorithmic categories. Columns represent subcategories. Table 5 in Appendix C summarizes the (sub)categories.

Table 4: Pass@1 (%) on OJBench using Claude-Sonnet-4.6. Direct generates code directly from the original requirements (OR) or requirements transformed by REFORM.

(a) Combination REFORM with reasoning and agents

Method	OR	REFORM	Δ
Direct	14.22	23.28	+9.06
Self-Planning	14.54	22.84	+8.30
SCoT	19.83	25.86	+6.03
OpenCode	29.31	32.33	+3.02

(b) Comparison REFORM with reasoning and agents

Method	Pass@1	Δ
REFORM	23.28	–
Self-Plan	14.54	−8.74
SCoT	19.83	−3.45

47.54%), yet their recommendations for category 8 are DR and ABR, respectively. Therefore, use preferences specific to the target model and problem type when choosing how to express a requirement; neither model performance nor problem category alone determines a suitable format.

6 SUITABLE REQUIREMENT FORMAT FOR REASONING AND AGENT

Reasoning strategies and coding agents change how a model proceeds from a requirement to code. We examine whether the choice of requirement format still matters when these downstream methods are used. On OJBench, we pair Claude-Sonnet-4.6 with direct generation, Self-Planning (Jiang et al., 2024), SCoT (Li et al., 2023), and OpenCode. For each method, Table 4 compares the original requirement (OR) with the requirement format selected by REFORM for the same task.

In Table 4(a), the REFORM-selected requirement raises Pass@1 under all four methods: direct generation from 14.22% to 23.28% (+9.06%), Self-Planning from 14.54% to 22.84% (+8.30%), SCoT from 19.83% to 25.86% (+6.03%), and OpenCode from 29.31% to 32.33% (+3.02%). Gains vary in size but persist with explicit planning, structured reasoning, and the coding agent.

Table 4(b) provides a complementary comparison: direct generation with REFORM reaches 23.28%, exceeding Self-Planning and SCoT with OR by 8.74% and 3.45%, respectively. Adding SCoT to the selected requirement raises Pass@1 to 25.86%, while OpenCode with the selected requirement achieves the highest score in panel (a), 32.33%. The improvements within each method and the stronger combined scores suggest that requirement-format selection complements both explicit reasoning and agentic code generation.

7 CONCLUSION

We systematically study how the requirement formats affect LLMs for code generation. Experimental results show that requirement formats are related to the requirement meaning and target model, with no single format consistently performing best. Motivated by these findings, we propose REFORM, which selects and reformulates requirement formats according to the target model and requirements. Evaluation demonstrates that REFORM outperforms the compared baselines. These findings highlight the importance of requirement formats and the value of adapting them to the model and task for code generation.

AI USE STATEMENT

In this work, we used generative AI tools to polish the writing and support literature retrieval. All AI-assisted text, literature suggestions, research ideas, and outputs were critically reviewed and verified by the authors. The authors independently checked the accuracy and relevance of suggested references and revised the resulting content as necessary. We take full responsibility for the final content of this work, including all text, claims, citations, and artifacts produced with the aid of generative AI.

REFERENCES

Alibaba Cloud. Qwen3.5-Plus Model Information. Alibaba Cloud Model Studio Documentation, 2026. URL https://help.aliyun.com/zh/model-studio/qwen3-5-plus. Accessed 2026-08-02.

Anthropic. Introducing Claude Sonnet 4.6. Anthropic, February 2026. URL https://www.anthropic.com/research/claude-sonnet-4-6. Accessed 2026-08-02.

Leonhard Applis, Yuntong Zhang, Shanchao Liang, Nan Jiang, Lin Tan, and Abhik Roychoudhury. Unified software engineering agent as ai software engineer. *arXiv preprint arXiv:2506.14683*, 2025.

Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2002.

Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.

DeepSeek-AI. DeepSeek V4 Preview Release. DeepSeek API Documentation, April 2026. URL https://api-docs.deepseek.com/news/news260424/. Accessed 2026-08-02.

John Doe. *Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)*. IEEE, New York, 2011.

Tulsee Doshi and Gemini Team. Gemini 3 Flash: Frontier Intelligence Built for Speed. Google, December 2025. URL https://blog.google/products-and-platforms/products/gemini/gemini-3-flash/. Accessed 2026-08-02.

Junbo Fu, Guoshuai Zhao, Yimin Deng, Yunqi Mi, and Xueming Qian. Learning to paraphrase for alignment with LLM preference. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 2394–2407, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.134. URL https://aclanthology.org/2024.findings-emnlp.134/.

Martin Glinz. Problems and deficiencies of uml as a requirements specification language. In *Tenth International Workshop on Software Specification and Design (IWSSD-10)*, pp. 11–22. IEEE, 2000. doi: 10.1109/IWSSD.2000.891122.

Sol Greenspan, John Mylopoulos, and Alex Borgida. On formal requirements modeling languages: Rml revisited. In *Proceedings of 16th International Conference on Software Engineering*, pp. 135–147. IEEE, 1994. doi: 10.1109/ICSE.1994.296773.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2308.00352.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer, 2011.

ISO/IEC/IEEE. ISO/IEC/IEEE international standard – systems and software engineering – life cycle processes – requirements engineering. ISO/IEC/IEEE 29148:2018(E), 2018. URL https://doi.org/10.1109/IEEESTD.2018.8559686.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination-free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Geonhui Jang, Dongyoon Han, and YoungJoon Yoo. StoryCoder: Narrative reformulation for structured reasoning in LLM code generation. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19092–19112. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.872. URL https://aclanthology.org/2026.acl-long.872/.

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

Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pp. 6766–6805, 2024.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. CodeChain: Towards modular code generation through chain of self-revisions with representative submodules. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=vYhglxSj8j.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 348–370. Springer, 2010.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*, 2023. URL https://arxiv.org/abs/2305.06599.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. The use and effectiveness of user stories in practice. In *Requirements Engineering: Foundation for Software Quality*, volume 9619 of *Lecture Notes in Computer Science*, pp. 205–222. Springer, 2016. doi: 10.1007/978-3-319-30282-9_14. URL https://doi.org/10.1007/978-3-319-30282-9_14.

Daniel Méndez Fernández, Stefan Wagner, Marcos Kalinowski, Michael Felderer, Priscilla Mafra, Antonio Vetrò, Tayana Conte, Marie-Therese Christiansson, Desmond Greer, Casper Lassenius, et al. Naming the pain in requirements engineering: Contemporary problems, causes, and effects in practice. *Empirical Software Engineering*, 22(5):2298–2338, 2017. doi: 10.1007/s10664-016-9451-7. URL https://doi.org/10.1007/s10664-016-9451-7.

Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

Alessandro Midolo, Alessandro Giagnorio, Fiorella Zampetti, Rosalia Tufano, Gabriele Bavota, and Massimiliano Di Penta. Guidelines to prompt large language models for code generation: An empirical characterization. In *Proceedings of the 2026 34th IEEE/ACM International Conference on Program Comprehension*, pp. 319–330, 2026.

Andrei Paleyes, Radzim Sendyka, Diana Robinson, Christian Cabrera, and Neil D. Lawrence. Prompt variability effects on LLM code generation. *arXiv preprint arXiv:2506.10204*, 2025. URL https://arxiv.org/abs/2506.10204v1.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Laboni Sarker, Mara Downing, Achintya Desai, and Tevfik Bultan. Assessing, exploiting, and mitigating syntactic robustness failures in LLM-based code generation. In *Proceedings of the 2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering*, 2026. doi: 10.1145/3793655.3793727. URL https://doi.org/10.1145/3793655.3793727.

John Ferguson Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Manning Publications, 2014. ISBN 9781617291654. URL https://www.manning.com/books/bdd-in-action.

Stack Overflow. 2025 developer survey: AI, 2025. URL https://survey.stackoverflow.co/2025/ai. Accessed 2026-09-21.

Alistair Sutcliffe. Scenario-based requirements engineering (SCRAM). In *User-Centred Requirements Engineering*, pp. 127–147. Springer, 2002. doi: 10.1007/978-1-4471-0217-5_6. URL https://doi.org/10.1007/978-1-4471-0217-5_6.

Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pp. 249–262, 2001. doi: 10.1109/ISRE.2001.948567. URL https://doi.org/10.1109/ISRE.2001.948567.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. ReCode: Robustness evaluation of code generation models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13818–13843, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.773. URL https://aclanthology.org/2023.acl-long.773/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025a. URL https://arxiv.org/abs/2407.16741.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. OJBench: A competition level code benchmark for large language models. *arXiv preprint arXiv:2506.16395*, 2025b. URL https://arxiv.org/abs/2506.16395.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990a. doi: 10.1109/2.58215. URL https://doi.org/10.1109/2.58215.

Jeannette M Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–22, 1990b.

Z.AI. GLM-5: Overview. Z.AI Developer Documentation, 2026. URL https://docs.z.ai/guides/llm/glm-5. Accessed 2026-08-02.

A SHARED PROMPT TEMPLATES

This appendix presents the shared prompt templates for requirement transformation and code generation. The individual requirement-format templates described in Section 2.1 appear in Appendix B. Typography and line wrapping are adjusted for readability.

In Prompt A.1, `[XXX]` denotes the selected requirement format, `[...]` is replaced with its template, and {`problem_statement`} is replaced with the original problem statement. In Prompt A.2, {`converted_requirement`} is replaced with the resulting requirement text. Both templates are sent as user messages.

Prompt A.1 Requirement transformation

```
# Role Definition

You are a programming-problem requirement reformulator. You excel at
converting natural language requirements into standardized [XXX]
format. Your task is to transform the original problem requirement
into the [XXX] requirement format.

# Rules:

* Preserve all important information from the original problem.
* Preserve the exact input format and output format.
* Preserve all constraints, examples, and edge cases.
* Do not solve the problem.
* Do not write code.
* Do not add assumptions that are not supported by the original
  statement.
* Write in English only.
* Use the exact structure below.

# [XXX] Requirement Format:

[...]

# Original Problem Requirement:

{problem_statement}
```

Prompt A.2 Code generation

```
# Role Definition

You are an expert competitive programming assistant. Write a correct
and efficient Python 3 solution for the programming problem described
below.

# Rules:

- Return only the final Python code.
- Do not include markdown code fences.
- Do not include explanations.
- Follow the input and output format exactly.
- Handle all edge cases.
- Use efficient algorithms suitable for the constraints.
- Do not use unsafe operations.
- Do not read or write files.
- Read from standard input and write to standard output.

# Problem Statement:

{converted_requirement}
```

B REQUIREMENT-FORMAT TEMPLATES

This appendix provides the 12 transformation templates described in Section 2.1. OR retains the original requirement and therefore has no transformation template. Template headings use the same names and abbreviations as the main text. The prompt bodies are reproduced with typographic normalization; they should be interpreted together with the shared transformation rules in Appendix A.

B.1 UNSTRUCTURED REQUIREMENT FORMATS

Template B.1.1 Narrative requirement (NR)

Story Genre: {Select from the available story genres.}

Story Title: {A short title describing the story scenario and task theme.}

Story Background: {Use a paragraph of natural language to describe the characters, system, organization, or scenario. Explain why this computation / judgment / construction problem arises.}

Task Overview: {Describe the task of the story character or the problem that needs to be solved. This corresponds to the core computational objective of the original problem.}

Input/Output Description: {Repackage the sample input and output as part of the story scenario flow, presenting it clearly and with context.}

Constraints: {Constraint 1} {Constraint 2}

Available Story Genres:

- Scheduling Story: Suitable for sorting, intervals, greedy algorithms, resource allocation, and task scheduling.
- Adventure Story: Suitable for graph search, path planning, dynamic programming, and state transitions.
- Management System Story: Suitable for databases, CRUD operations, APIs, new feature requests in codebases, and state maintenance.
- Game Rule Story: Suitable for game theory, simulation, state machines, and combinatorial construction.
- Detective Reasoning Story: Suitable for constraint satisfaction, truth/falsehood judgment, and inferring hidden information.
- Logistics and Transportation Story: Suitable for shortest paths, flow, matching, and capacity constraints.

Rules:

- The story must serve the requirements.
- Do not add irrelevant plotlines.
- Every key object in the story must map back to a variable, data structure, or module in the original problem.
- Do not alter the problem's rules or constraints due to the storytelling.
- The recommended narrative order is: Background → Problem → Information → Rules → Objective → Constraints.
- The mechanical feel of input/output formats can be softened, but the information necessary for implementation must be retained.
- For algorithmic problems, the story must not replace constraint specifications; constraints must still be listed separately.

Template B.1.2 Pain-point description (PPD)

Current Pain Point: {Describe the difficulty currently encountered by the user, developer, or system.}

Impact of the Pain Point: {Explain the problems caused by this difficulty, such as low efficiency, high error rates, inability to handle large-scale data, inconsistent results, etc.}

Desired Capability: A {program / function / module / interface} needs to be implemented that can automatically complete {core functionality} based on {input data / parameters / current system state}.

Pain Point Resolution Criteria:

- Users no longer need to manually perform {the originally difficult operation}.
- The program can stably process all data within the specified problem range.
- The program results are consistent with the rule definitions.
- The implementation must not break existing interfaces, semantics, or compatibility requirements.

Constraints: {Data ranges, complexity requirements, error handling, invariants, global constraints, etc.}

B.2 Semi-Structured Requirement Formats

Template B.2.1 User story (US)

User Story: As a {role}, I want {the core functionality to be completed by the system / function / program} so that {the value or outcome the user can obtain}.

Scenario: An instance of behavior under specific conditions for a user story. Each story includes at least one scenario covering normal flows, exception branches, and edge cases.

Structured format:

- Scenario: <Scenario name, summarizing the test objective>
- Given <Precondition 1> [And <Precondition n>]
- When <Interaction action>
- Then <Expected outcome 1> [And <Expected outcome m>]

Given: The initial state of the system before entering the scenario.

When: The user/system action that triggers the scenario.

Then: The observable changes in the system state after the action.

Acceptance Criteria:

Quantifiable, unambiguous validation rules derived from scenarios; the ultimate basis for determining whether a user story is complete.

- If a scenario has no additional variables, the acceptance criterion is directly equivalent to the expected outcome in the Then clause.
- If a scenario includes parameterized examples, each combination of parameters corresponds to an acceptance criterion.

Template B.2.2 Dimensional Requirements (DR)

Requirement Background: Describe the background and business scenario of the requirement, providing relevant contextual knowledge for code implementation.

Requirement Purpose: Describe the core functionality of the requirement, summarizing the intended goal or task to be achieved.

Terminology Explanation: Explain domain terms, symbols, and specialized concepts involved in the requirement to enhance comprehensibility.

Input Requirements: Detail the input needed by the code, including input types, formats, and any other necessary constraints.

Output Requirements: Describe the code's output, including output types, formats, and any other necessary constraints.

Example Explanation: Explain the input and output of the examples provided in the requirement.

Edge Cases: Describe the boundary conditions and special input scenarios of the requirement.

Key Functions: Highlight key functionalities and error-prone features, emphasizing information that significantly impacts code correctness.

API: Specify external APIs and library functions involved in the implementation, describing their usage and limitations.

Error Handling Requirements: Describe the code's behavior and impact under exceptional conditions, such as invalid input or call failures.

Invariants: Describe properties or conditions that must remain unchanged throughout code execution.

Global Constraints: Describe restrictions the code must satisfy, such as complexity, resource usage, and data scale.

Hints or Tips: Provide supplementary information helpful for understanding the requirement, which does not constitute mandatory functional constraints.

Template B.2.3 Scenario-driven requirement (SDR)

Scenario Name: {Describe the usage scenario corresponding to this code generation task}

System Context Model / Structure Model:

- **System:**{The program, function, module, or codebase to be generated.}
- **Agents / External Entities:**
  - User / Caller: {The person or external program that invokes the program, function, or system.}
  - Input Source: {Standard input, function parameters, files, CLI arguments, existing codebase interfaces, etc.}
  - Output Receiver: {Standard output, return values, files, test framework, caller, etc.}
  - External Component: {e.g., existing codebase, library functions, runtime environment; omit if none.}
- **System Boundary:**
  - Initial Context: {Initial conditions before execution, e.g., input already provided, certain interfaces already exist in the codebase, target language already determined.}
  - Input Event: {The input event entering the system, e.g., reading input, function call, receiving CLI arguments, invoking existing API.}

- System Process: {The main computation, judgment, state update, or codebase modification the system needs to perform.}
- System Output: {The output produced by the system, e.g., return value, printed content, file changes, codebase behavioral changes.}
- Output Destination: {To whom or where the output is delivered, e.g., test cases, caller, terminal, target file, existing module.}
- Acceptability Condition: {Under what conditions the output is considered acceptable, e.g., passing tests, behavior consistent with reference program, preserving semantics, etc.}

Goal-Function Correspondence Check:

- User Goal: {The goal the user or test aims to achieve}
- System Function:{The function the generated code needs to provide}
- Correspondence:{How the function satisfies the user goal}

Event-Process Correspondence Check:

- Input Event: {Input event}
- Required Process:{The processing the system must perform}
- Coverage:{Whether the input event is covered by the processing logic}

Output-Destination Correspondence Check:

- System Output:{Output content}
- Destination:{Output receiver}
- Acceptance:{Whether the output meets the requirements of the receiver or test framework}

Uncovered Areas or Risk Points:

- Whether any input event is unhandled.
- Whether any output event is unhandled.
- Whether any user goal lacks a corresponding system function.
- Whether output conflicts exist under different scenarios.

Template B.2.4 Layered requirement (LR)

Requirement Title / Title: {No more than approximately 5 words, summarizing the requirement}

Requirement Description / Description: {Describe the capabilities that the program, function, module, or codebase to be generated should possess}

Reason / Benefit / Rationale: {Explain why this requirement is needed or what task goal it serves}

Restrictions / Risks: {Describe the restrictions, risks, boundaries, or uncertainties of this requirement}

Requirements Abstraction Level:

- Product Level: {The highest-level goal: what capability the entire program, system, or codebase ultimately provides}
- Feature Level: {What observable characteristics or user-perceptible capabilities the system must support}
- Function Level: {What specific functions the system, function, or module must perform; should be as testable and unambiguous as possible}

- Component Level: {Which components, interfaces, files, modules, dependencies, or technical constraints are involved in implementing this function}

Work-up Relation:

- Upper-Level Requirement: {Which Product / Feature level goal the current requirement supports}
- Current Requirement: {The Feature / Function / Component requirement currently being described}
- Lower-Level Decomposition: {Functional or component requirements into which this requirement can be further decomposed}

Code Generation Task Mapping:

- Generation Target:
  [ ] Function
  [ ] Script / Program
  [ ] Repository Change
  [ ] Full Codebase
  [ ] Translated Program
- Input Source: {Natural language problem statement, existing codebase, program documentation, binary behavior, source language code, etc.}
- Output Form: {Function implementation, complete program, patch, complete codebase, target language code, etc.}

Testability Statement: {Describe how this requirement can be tested, e.g., unit tests, hidden tests, end-to-end tests, behavioral consistency tests}

Template B.2.5 Goal-driven requirement (GDR)

Root Goal:

- G0 {The ultimate goal that the generated code should satisfy}
- Goal Type
  [ ] Achieve
  [ ] Maintain
  [ ] Avoid
  [ ] Optimize

Goal Specification: {Describe the goal in a declarative manner, rather than directly writing implementation steps}

Goal Motivation: {Why this goal needs to be satisfied; corresponds to the problem objective, user requirement, or benchmark evaluation goal}

Agent Assignment:

- Software-to-be {The responsibilities that the generated code needs to assume}
- Environment {The responsibilities assumed by the input, test framework, existing codebase, or runtime environment}
- External Component {Third-party libraries, existing modules, compilers, interpreters, etc.; omit if none}

Goal Refinement:

- Refinement Type [ ] AND-refinement [ ] OR-refinement
- G1 {Input reception or interface adaptation goal}
- G2 {Core data representation or state modeling goal}
- G3 {Main computation, transformation, judgment, or codebase behavior goal}
- G4 {Edge case, exception, or special state handling goal}

- G5 {Performance, resource, language, dependency, or compatibility goal}
- G6 {Output format, return value, patch form, or behavioral consistency goal}
- Refinement Relation G0 is satisfied by G1 ∧ G2 ∧ G3 ∧ G4 ∧ G5 ∧ G6.

Operationalization:

- Operation O1 {The operation that realizes G1}
  - Required Precondition {Precondition}
  - Required Trigger Condition {Trigger condition}
  - Required Postcondition {Postcondition}
- Operation O2 {The operation that realizes G2/G3}
  - Required Precondition {...}
  - Required Trigger Condition {...}
  - Required Postcondition {...}
- Operation O3 {The operation that realizes G6}
  - Required Precondition {...}
  - Required Trigger Condition {...}
  - Required Postcondition {...}

Conflict:

- Conflict C1 {The conflict between one goal and another}
- Conflict Resolution {How to resolve the conflict, e.g., prioritize correctness, then performance; or specify output selection rules}

Obstacles:

- O1 {A situation that may hinder the achievement of G0/Gi, e.g., excessively large input scale, ambiguous requirements, incompatible existing interfaces}
- O2 {Another obstacle}
- Obstacle Resolution {Resolve obstacles by adding new sub-goals, replacing agents, modifying operationalization methods, etc.}

Goal Satisfaction Check:

- G1 satisfied if: {Input or interface goal satisfaction condition}
- G2 satisfied if: {Data representation or state goal satisfaction condition}
- G3 satisfied if: {Core functionality goal satisfaction condition}
- G4 satisfied if: {Boundary or exception goal satisfaction condition}
- G5 satisfied if: {Performance or constraint goal satisfaction condition}
- G6 satisfied if: {Output or behavioral consistency goal satisfaction condition}

Template B.2.6 Behavior-driven requirement (BDR)

Feature: {The functionality the problem requires}

As a developer,

I want {description of the core functionality to be implemented},

So that {satisfy the problem's requirements / time complexity / robustness goals}

Background:

Given the input is {input data type and structure}

And the input range is {data constraints and boundary conditions}

Scenario: Normal scenario examples.

Given the input = <normal input sample>

When the target algorithm is executed

Then return <expected output result>

Scenario Outline: Batch boundary and exception scenarios

Given the input = <input>

When the target algorithm is executed

Then return <output>

Examples:

input	output
boundary input 1	expected result 1
boundary input 2	expected result 2
exception input 1	expected result 3
exception input 2	expected result 4

Template B.2.7 Test-driven requirement (TDR)

Functional Goal:

- Implement:[Function Name]
- Input:[Data type / format / range]
- Output:[Data type / format / judgment criteria]

Constraints and Rules: {Describe the business rules, algorithmic rules, data ranges, and special handling requirements that must be followed.}

Test Cases:

- When the input is {normal input sample 1}, the system should return {expected output result 1}.
- When the input is {normal input sample 2}, the system should return {expected output result 2}.
- When the input is {boundary input sample 1}, the system should return {expected boundary result 1}.
- When the input is {boundary input sample 2}, the system should return {expected boundary result 2}.
- When the input is {exceptional / invalid input sample 1}, the system should return {expected exception result 1}.
- When the input is {exceptional / invalid input sample 2}, the system should return {expected exception result 2}.

B.3 Structured Requirement Formats

Template B.3.1 Automaton-based requirement (ABR)

Function: {The core function required by the problem}

State Set: {List all states required during problem solving}

Initial State: {The start state of the automaton}

Terminal State: {The state where execution terminates and the answer is output}

Input Alphabet: {All valid input types / values specified by the problem}

Output Alphabet: {All valid output types / values specified by the problem}

State Transition Rules:

Current State 1 + [Input 1] → [Next State 1], Output [Result 1]

Current State 2 + [Input 2] → [Next State 2], Output [Result 2]
(Complete all transitions following the logic of the problem)

Input Format: {Precise input format required by the problem}

Output Format: {Precise output format required by the problem}

Constraints: {Data range, time limit, special restrictions}

Normal Example:

- Input: {Sample input}
- Output: {Corresponding sample output}

Template B.3.2 Contract-based requirement (CBR)

Task: {Describe the functionality to be implemented.}

Rules:

- **Preconditions:** Requirements that must be satisfied before execution; conditions assumed to hold by default.
- **Postconditions:** The result state that must be guaranteed after the method successfully executes, given that the preconditions are satisfied.
- **Invariants:** Conditions that must remain true throughout the entire execution; the invariant state must not be violated.

Preconditions:

The input format satisfies {input type requirements}

Data structure properties: {constraints on length, range, connectivity, element values, etc.}

Postconditions:

The output must satisfy {result correctness conditions}

The output must satisfy {mathematical relationship / logical relationship} with respect to the input

The output format satisfies {output format requirements}

Invariants:

Throughout execution, {core data structure properties} always remain unchanged

{Global constraints / state validity} always hold true throughout the entire execution

Template B.3.3 Formal specification (FS)

Domain Knowledge:

The core constructs of a formal specification include:

- **requires:** The precondition that must be satisfied before function execution {input constraints}
- **ensures:** The postcondition that must be satisfied after function execution completes {output/state constraints}
- **modifies:** The variables or memory locations the function is allowed to modify {side-effect constraints}
- **loop invariant:** The invariant that must hold true before each iteration of a loop {loop behavior constraints}

Formal specifications support standard logical syntax:

- Logical operators: '&&' (and), '||' (or), '!' (not), '==>' (implication), '<==>' (bidirectional implication)
- Quantifiers: 'forall' (universal quantifier), 'exists' (existential quantifier)
- '::': Quantifier-specific delimiter, used to separate quantifier variables from constraint predicates

Rules for generating formal language:

- Completely cover all functionality and constraints of the problem, with no omissions
- The specification must be unambiguous, allowing only a single interpretation
- All logical symbols, quantifiers, and syntax formats must be used correctly to ensure rigorous and verifiable specifications

Output format requirements: You must strictly output the final content in the following format:

{

- Output Requirements:
  {The function implementation, complete program, patch, complete codebase, target language code, etc.required by the problem}
- Formal Language:
  - requires ...
  - ensures ...
  - modifies ...
  - loop invariant ...

}

C ALGORITHM CATEGORIES AND CLASSIFICATION PROMPT

This appendix summarizes the algorithm categories shown in Figure 4 and presents the complete algorithm classification prompt used in Section 4.1.

Table 5: Algorithm categories and brief descriptions.

Category	Subcategory	Brief description
1 Dynamic Programming	1.1 Process-Based Problems	Reuse state transitions across stages of a process.
	1.2 Structure-Based Problems	Reuse subproblems over structured data and dependent decisions.
2 Greedy Algorithms	—	Optimize through successive locally optimal choices.
3 Mathematics	3.1 Integer Properties	Analyze divisibility, primes, and modular relationships.
	3.2 Numerical Representations	Work with digits, numeral bases, and numerical forms.
	3.3 Algebraic Derivation	Derive answers from formulas and algebraic relationships.
4 Combinatorics	4.1 Probability, Expectation, and General Counting Principles	Analyze random outcomes or apply general counting principles.
	4.2 Ordered Arrangement Counting	Count permutations and other ordered arrangements.
	4.3 Unordered Configuration Counting	Count selections and configurations without regard to order.
5 Strings	5.1 Pattern Matching	Identify patterns and matching relationships in strings.
	5.2 Editing and Transformation	Edit strings to reach a target state.
	5.3 Algorithmic Applications	Scan, map, parse, or construct character sequences.
6 Graphs and Trees	6.1 Trees	Analyze tree structures and relationships.
	6.2 Graph Optimization	Compute numerical or optimal results on graphs.
	6.3 Graph Structural Properties	Analyze graph connectivity, traversal, and structural relationships.
7 Geometry	7.1 Grid Construction	Arrange cells and compute grid coverage or placement.
	7.2 Shape Calculations	Compute shape measurements using geometric formulas.
	7.3 Point and Line Calculations	Analyze coordinate relationships among points, lines, and vectors.
8 Data Structures	—	Maintain data structures or process online updates and queries.
9 Implementation	9.1 Process Execution	Simulate a deterministic process.
	9.2 Rule Checking	Check whether inputs satisfy fixed rules.
	9.3 Direct Conversion	Perform direct input, output, or representation conversions.
10 Constructive Algorithms	—	Construct any object satisfying the given constraints.

Continued on next page

Table 5 (continued)

Category	Subcategory	Brief description
11 Binary Search on the Answer	—	Find an extremal answer through monotonic feasibility checks.
12 Two Pointers and Sliding Windows	—	Maintain contiguous intervals by moving their endpoints.
13 Sorting and Searching	—	Order elements or retrieve them by rank or value.
14 Game Theory	14.1 Impartial Combinatorial Games	Determine outcomes of alternating object-removal games.
	14.2 Adversarial Strategy Games	Determine outcomes or strategies under optimal play.
15 Bit Manipulation	—	Manipulate or inspect individual bits of integers.
16 Prefix Sums and Hashing	—	Aggregate prefix or frequency information for computation.
17 Other	—	Problems outside all other categories.

Prompt C.1 Algorithm classification

 # Algorithm Category Definitions

General Classification Rules

1. Every problem must be assigned to exactly one category.
2. If a problem statement satisfies the rules for multiple categories, select the first eligible category in the priority order given below.
3. For a subcategory, the problem must first satisfy the entry criteria for its parent category; then select exactly one subcategory within that parent category.
4. *Other* applies only to problems that satisfy none of the preceding category rules.

Category Priority and Definitions

15 Bit Manipulation

Entry criteria. The statement requires operations on the binary representations of integers: XOR, bitwise AND, bitwise OR, shifts, bitmasks, setting or clearing bits, counting bits, or comparing bits.

Exclusion criteria. Ordinary binary input/output, parity checks, or exponentiation alone do not qualify.

14.1 Game Theory / Impartial Combinatorial Games (Symmetric Combinatorial Games)

Entry criteria. There are at least two players who take turns removing objects from heaps or collections of stones, coins, or items. Winning or losing is determined by the inability to move, the last move, the number of remaining objects, an XOR value, Nim, or Grundy values.

Exclusion criteria. This category does not apply unless at least two players take turns acting.

14.2 Game Theory / Adversarial Strategy Games (Asymmetric Combinatorial Games)

Entry criteria. There are at least two players, and the task asks for the winner, loser, score, draw outcome, optimal strategy, or final state under optimal play.

Exclusion criteria. Merely simulating fixed rules, with no choice of optimal actions by players, does not qualify.

6.1 Graphs and Trees / Trees

Entry criteria. The input is a tree, or the statement uses any of the following concepts: root, parent, child, ancestor, descendant, subtree, leaf, or path in a tree.

Exclusion criteria. The word ‘‘tree’’ referring to a physical object without node--edge relationships does not qualify.

6.2 Graphs and Trees / Graph Optimization

Entry criteria. The input is a graph, and the task requires numerical computation or optimization on its structure. This includes path lengths, shortest or longest paths, distances, costs, capacities, rewards, probabilities, times, flows, cuts, matchings, spanning trees, weighted connectivity configurations, or other graph results that depend on edge weights, vertex weights, path weights, or connection costs.

Exclusion criteria. If the graph has no edge weights, costs, capacities, or path costs, and the core task concerns connectivity, reachability, cycles, bipartiteness, topological relationships, structural properties, unweighted traversal, or the minimum number of

Prompt C.1 Algorithm classification

unweighted steps, assign it to 6.3. Trees and tasks using tree-structure concepts take priority under 6.1.

6.3 Graphs and Trees / Graph Structural Properties (Graph Topology)

Entry criteria. The input is a graph, and the core task is to determine or analyze structural properties: connectivity, reachability, connected components, cycles, bipartiteness, graph coloring, topological dependencies, minimum unweighted steps, BFS levels, DFS traversal, Eulerian paths, Hamiltonian paths, degree relationships, structural validity, or relationships involving edge or vertex insertion and deletion.

Exclusion criteria. If the core task depends on edge or vertex weights, costs, capacities, path costs, weighted optimization, flows, cuts, matchings, spanning trees, or other numerical graph computations, assign it to 6.2. Trees and tasks using tree-structure concepts take priority under 6.1.

7.1 Geometry / Grid Construction (Grid Layout)

Entry criteria. The objects are grids, boards, tables, square cells, hexagonal grids, lattice points, or tiles. The task requires computing layouts, coverage, adjacency, placement, area measured in cells, or grid coordinates.

Exclusion criteria. Movement reachability, shortest paths, and graph traversal belong to Graphs and Trees. Path counting or optimal paths belong to Dynamic Programming when state transitions are central.

7.2 Geometry / Shape Calculations (Geometric Formula Applications)

Entry criteria. The objects or concepts include circles, radii, diameters, triangles, polygons, rectangles, squares, angles, areas, perimeters, volumes, surface areas, convexity, similarity, congruence, Pythagorean relationships, distance formulas, or geometric solids. The answer primarily follows from geometric formulas, shape measurements, geometric relationships, or algebraic derivations of geometric properties.

Exclusion criteria. Array indices or table coordinates alone do not qualify. If the core task concerns computational geometry involving coordinate points, lines, vectors, directions, cross products, intersections, or convex hulls, assign it to 7.3.

7.3 Geometry / Point and Line Calculations (Spatial Vector Calculations)

Entry criteria. The objects or concepts are points, lines, line segments, rays, vectors, coordinate axes, slopes, directions, cross products, dot products, collinearity, intersections, relative positions, convex hulls, nearest points, projections, rotations, or reflections in the plane or in space. The task requires computing or determining coordinate relationships among these objects.

Exclusion criteria. If vertices and edges form a graph and the task concerns graph connectivity, paths, or tree structures, assign it to Graphs and Trees. If coordinates merely denote grid or board positions and the core task concerns cell layout, coverage, or placement, assign it to 7.1.

5.1 Strings / Pattern Matching (Pattern Recognition)

Entry criteria. The input consists of strings, and the task concerns substrings, subsequences, pattern matching, palindromes, anagrams, regular expressions, wildcard matching, word matching, prefix functions, Z-functions, KMP, trie matching, rolling-hash matching, string search, or text pattern recognition.

Exclusion criteria. Strings used only as an input format do not qualify if character sequences are not the objects being processed.

Prompt C.1 Algorithm classification

If the core task transforms a string into a target state through editing operations, assign it to 5.2. If it uses the linear sequence structure of strings for traversal, counting, construction, mapping, compression, parsing, or output, assign it to 5.3.

5.2 Strings / Editing and Transformation (State Editing and Transformation)

Entry criteria. The input consists of strings. The task transforms a string into a target state through replacement, insertion, deletion, movement, swapping, appending, reversal, rearrangement, keyboard input, case conversion, or encoding conversion; alternatively, it asks for the number or cost of transformations, the minimum number of operations, or reachability.

Exclusion criteria. This category does not apply without string transformation operations. If the core task concerns pattern matching, substrings, palindromes, or prefix/suffix relationships, assign it to 5.1. Linear traversal, counting, construction, mapping, compression, parsing, or format validation belongs to 5.3.

5.3 Strings / Algorithmic Applications (Linear Scanning and Mapping)

Entry criteria. The input consists of strings, and the task uses their linear sequence structure for traversal, scanning, splitting, concatenation, sorting, comparison, character-frequency counting, character mapping, compression, encoding or decoding, text-structure parsing, format validation, string construction, or output of strings satisfying character constraints.

Exclusion criteria. Substring or subsequence matching, pattern recognition, prefixes/suffixes, palindromes, and regular-expression matching belong to 5.1 when central to the task. Editing, replacement, insertion, deletion, swapping, movement, and other transformation operations belong to 5.2 when central to the task.

8 Data Structures

Entry criteria. The statement explicitly requires using or maintaining a stack, queue, deque, heap, priority queue, disjoint-set union, segment tree, Fenwick tree, ordered set, multiset, monotonic stack, or monotonic queue; alternatively, it requires online processing of updates and queries.

Exclusion criteria. A one-time offline aggregation without maintaining dynamic state does not qualify.

11 Binary Search on the Answer

Entry criteria. The task asks for the minimum feasible value, maximum feasible value, minimization of a maximum, or maximization of a minimum, with a monotonic structure supporting feasibility checks on candidate answers.

Exclusion criteria. Merely searching for an element in a sorted array belongs to Sorting and Searching. Merely mentioning binary representations does not qualify.

12 Two Pointers and Sliding Windows (Interval Scanning)

Entry criteria. The objects are contiguous intervals of an array or string. The task asks for the longest, shortest, or optimal interval, or the number of such intervals, and constraints can be maintained by moving the left and right endpoints.

Exclusion criteria. Arbitrary subsequences, noncontiguous sets, and paths in trees do not qualify.

16 Prefix Sums and Hashing (State Accumulation)

Entry criteria. The task requires range sums, prefix sums, difference arrays, cumulative quantities, frequency tables,

Prompt C.1 Algorithm classification

hash-table counting, rolling hashes, or prefix XOR. The answer depends on combining prefix or frequency states.

Exclusion criteria. String prefixes used as textual relationships, without hashing or cumulative quantities, belong to Strings.

1.1 Dynamic Programming / Process-Based Problems

Entry criteria. The task asks for a minimum, maximum, number of solutions, feasibility, probability, or expectation over a grid, board, state machine, sequence of operations, sequence of rounds, path process, or finite state space. The current state is obtained through transitions from the preceding stage or predecessor states; the overall computation advances as a simulated process and requires reusing subproblem results.

Exclusion criteria. A single simulation under deterministic rules, without state transitions and subproblem reuse, belongs to 9.1, Implementation / Process Execution. General dynamic programming over sequences, arrays, intervals, sets, trees, directed acyclic graphs (DAGs), digits, or resource states belongs to 1.2.

1.2 Dynamic Programming / Structure-Based Problems

Entry criteria. The task asks for a minimum, maximum, number of solutions, feasibility, probability, or expectation over sequences, arrays, intervals, sets, subset states, digits, trees, DAGs, dependencies, resource states, or general discrete state spaces. The answer requires state definitions, state transitions, and subproblem reuse, with multiple interacting decisions or dependencies.

Exclusion criteria. Local choices at each step without subproblem reuse belong to Greedy Algorithms. A single deterministic simulation belongs to 9.1. Answers obtained directly from closed-form formulas, number-theoretic properties, or simple algebraic relationships belong to Mathematics. Deterministic combinatorial counting without state transitions belongs to Combinatorics.

4.1 Combinatorics / Probability, Expectation, and General Counting Principles (Random Distribution Analysis)

Entry criteria. The statement explicitly asks for a probability, expectation, outcome of a random process, or distribution or expected value after random selection. It also includes formula-based counting, recurrence-based counting, evaluation of counting expressions, inclusion--exclusion, generating functions, Catalan-type formulas, and the pigeonhole principle, provided that ordered arrangements or unordered selection configurations are not the central counting objects.

Exclusion criteria. Permutations, rankings, relative order, and counts of rearrangements or ordered arrangements belong to 4.2. Subsets, combinations, partitions, pairings, selections, groupings, colorings, and configuration counts belong to 4.3. Tasks requiring state transitions and subproblem reuse belong to Dynamic Programming.

4.2 Combinatorics / Ordered Arrangement Counting (Permutation Counting)

Entry criteria. The task asks to compute permutations, rankings, relative order, counts of permutations, rearrangements, or ordered arrangements, or the number of permutation objects satisfying ordering constraints.

Exclusion criteria. Merely sorting elements and outputting them does not qualify. Selecting elements without regard to order belongs to 4.3. Producing any one permutation satisfying constraints may belong to Constructive Algorithms according to the priority rules when construction, rather than counting, is central.

Prompt C.1 Algorithm classification

4.3 Combinatorics / Unordered Configuration Counting (Combination Counting)

Entry criteria. The task asks for counts of subsets, combinations, partitions, pairings, selections, groupings, colorings, item configurations, resource allocations, or discrete objects satisfying constraints. The central objects involve selection, allocation, pairing, grouping, or configuration without emphasizing order.

Exclusion criteria. Counting graph paths or string substrings belongs to the corresponding graph or string category. Permutation order, rankings, and ordered arrangements belong to 4.2. Explicit requests for probabilities, expectations, or random processes belong to 4.1. Tasks requiring state transitions and subproblem reuse belong to Dynamic Programming.

3.1 Mathematics / Integer Properties

Entry criteria. The task concerns modular arithmetic, remainders, divisibility, factors, multiples, primes, composite numbers, greatest common divisors, least common multiples, congruences, Euler's totient function, modular inverses, prime factorization, divisor counts, divisor sums, or number-theoretic functions.

Exclusion criteria. A modulus used only for reporting the output does not qualify when dynamic programming or combinatorial counting is central. Tasks concerning only digits, numeral bases, fractions, decimals, powers, or numerical representations belong to 3.2.

3.2 Mathematics / Numerical Representations

Entry criteria. The task concerns digits, decimal or binary representations, representations in arbitrary bases, base conversion, digit counts, digit sums, digit rearrangement, fractions, decimals, percentages, scientific notation, precision, rounding, powers, exponents, logarithms, or conversion between numeric strings and numerical values.

Exclusion criteria. XOR, bitwise AND/OR, shifts, bitmasks, setting or clearing bits, counting bits, and comparing bits belong to Bit Manipulation. If divisibility, primes, factors, multiples, or modular arithmetic is central, assign it to 3.1.

3.3 Mathematics / Algebraic Derivation

Entry criteria. The answer follows from closed-form expressions, equations, inequalities, parity, floor or ceiling operations, simple algebraic relationships, sequence patterns, ratios, averages, sum formulas, difference formulas, boundary formulas, or direct mathematical derivations.

Exclusion criteria. Global optimization over multiple choices requiring state transitions belongs to Dynamic Programming. Number-theoretic properties belong to 3.1. Digits, numeral bases, fractions, decimals, powers, exponents, and numerical representations belong to 3.2. Counting discrete solutions belongs to Combinatorics.

2 Greedy Algorithms

Entry criteria. The task requires selecting, scheduling, deleting, merging, or allocating objects to optimize an outcome. Each choice can be determined by the current ordering, minimum, maximum, deadline, or available resources, without backtracking over states.

Exclusion criteria. Choices that depend on multiple historical states and require subproblem reuse belong to Dynamic Programming.

13 Sorting and Searching (Sequence Order and Retrieval)

Entry criteria. The core task concerns sorting, stable sorting, custom comparison, ranking, the k-th smallest or largest element,

Prompt C.1 Algorithm classification

medians, binary search in a sorted collection, or merging or retrieving elements in order.

Exclusion criteria. Sorting used only as preparation for greedy algorithms, binary search on the answer, or interval scanning belongs to the corresponding category.

10 Constructive Algorithms

Entry criteria. The task asks for any object satisfying the constraints, or for constructing a permutation, array, string, graph, matrix, or operation sequence after determining feasibility.

Exclusion criteria. Tasks requiring only a numerical answer without constructing an object do not qualify.

9.1 Implementation / Process Execution

Entry criteria. The statement specifies a deterministic process, events, rounds, time progression, state updates, or repeated operations. The task executes these rules and outputs the final state or intermediate results.

Exclusion criteria. Processes involving optimal decisions belong to Game Theory, Greedy Algorithms, or Dynamic Programming.

9.2 Implementation / Rule Checking

Entry criteria. The statement specifies fixed rules. The task checks input validity, set membership, or whether a state satisfies conditions, or selects objects that satisfy the rules.

Exclusion criteria. Constructing a new object satisfying the rules belongs to Constructive Algorithms.

9.3 Implementation / Direct Conversion

Entry criteria. The task only requires parsing input, applying a direct formula, conditional branching, formatting output, unit conversion, or directly rewriting strings or numbers.

Exclusion criteria. This category does not apply if any preceding algorithm category rule is satisfied.

17 Other

Entry criteria. The problem statement satisfies none of the entry criteria above.

Exclusion criteria. This category does not apply if any preceding entry criterion is satisfied.