Coding Assignment Evaluation Rubric

*Assessing Functional Performance **and** Conceptual Understanding*

Purpose & Scope

This rubric evaluates student coding submissions with a strong emphasis on **demonstrated conceptual understanding**, not just whether the code runs. It is designed for use in hybrid human + AI workflows (Claude, Gemini, or MiniMax) and works best when students submit both code *and* a written explanation/reflection.

Total Points: 100 (weights are suggestions — customize per assignment)

How to Use This Rubric with LLMs (Claude / Gemini / MiniMax)

1. Paste the **full assignment description**, learning objectives, and any starter code or test cases.
2. Provide this rubric (or the condensed version at the end).
3. Ask the model to: (a) quote specific evidence from the student's code/comments/explanation for each score, (b) output structured JSON or clear sections, and (c) flag borderline or surprising submissions for human review.
4. **Always** have a human teacher do final review — especially for high-stakes grades and nuanced understanding.

Recommended workflow: MiniMax or Gemini Flash for high-volume first pass → Claude for deeper analysis of complex or borderline work.

Suggested Point Weights (Adjust to Your Goals)

Criterion	Points	Focus
1. Functional Correctness & Robustness	30	Does it work? Edge cases? Error handling?
2. Demonstration of Conceptual Understanding ★	25	Core emphasis — evidence of deep grasp
3. Code Quality, Readability & Maintainability	20	Clean, professional code
4. Design, Modularity & Algorithmic Choices	15	Appropriate structures & trade-offs
5. Documentation, Comments & Reflective Explanation	10	Communication of thinking process

★ The Conceptual Understanding criterion is deliberately weighted high because working code alone does not prove understanding.

1. Functional Correctness & Robustness (30 points)

Measures whether the program correctly solves the stated problem, handles edge cases, and behaves robustly under unexpected input.

Excellent (27–30 pts)

- All core functionality works perfectly on provided tests and additional edge cases
- Graceful error handling and input validation (no crashes on bad input)
- Clear evidence of defensive programming and testing mindset
- Edge cases explicitly considered and handled appropriately

Good (22–26 pts)

- Core functionality mostly correct; minor bugs on rare edge cases
- Some error handling present but incomplete
- Passes most tests; fails gracefully on some invalid input

Satisfactory (15–21 pts)

- Basic functionality works for normal cases but breaks on several edge cases
- Little or no error handling; may crash on invalid input
- Significant bugs that affect usability

Needs Improvement (0–14 pts)

- Major functionality broken or incomplete
- Frequent crashes or incorrect output on normal inputs
- Little evidence of testing or robustness consideration

2. Demonstration of Conceptual Understanding (25 points) ★

Most important criterion for this rubric. Evaluates whether the student shows genuine understanding of the underlying concepts, not just the ability to produce working code (which can be achieved via AI assistance or trial-and-error). Look for evidence in both the code structure/naming/comments *and* the written explanation.

Excellent (22–25 pts)

- Written explanation accurately and clearly explains key concepts and why specific choices were made
- Code structure, naming, and organization directly reflect taught principles (e.g., proper encapsulation, correct use of recursion/iteration, appropriate data structures)
- Student identifies trade-offs, alternatives considered, or limitations of their approach
- Comments explain *why* (not just *what*) and connect to course material
- Evidence that student could explain or extend the solution in a viva or follow-up question

Good (17–21 pts)

- Explanation is mostly accurate with minor gaps or superficial parts
- Code shows good application of concepts but some choices feel mechanical rather than deeply understood
- Some evidence of reasoning in comments or explanation, but not fully developed
- Student can likely explain most of their work

Satisfactory (10–16 pts)

- Explanation describes *what* was done more than *why*; some conceptual inaccuracies
- Code works but shows limited grasp (e.g., uses correct structure by coincidence or copy-paste patterns)
- Few or vague connections to taught concepts; minimal justification of design choices

Needs Improvement (0–9 pts)

- Little or no meaningful explanation of concepts or design decisions
- Code appears to be trial-and-error or heavily AI-generated without understanding
- Fundamental misconceptions visible in implementation or explanation
- Cannot articulate why the solution works or what the key concepts are

3. Code Quality, Readability & Maintainability (20 points)

Assesses professional coding practices: clarity, consistency, naming, structure, and long-term maintainability.

Excellent (18–20 pts)

- Excellent, consistent naming that clearly reflects purpose and domain concepts
- Clean, logical structure with appropriate functions/classes/modules
- Follows language idioms and style conventions consistently
- Minimal duplication; easy for another developer to read and modify

Good (14–17 pts)

- Generally good naming and structure with minor inconsistencies
- Some functions may be long or do too much, but overall readable
- Mostly follows style guidelines

Satisfactory (9–13 pts)

- Variable/function names are vague or inconsistent
- Large functions or classes with mixed responsibilities
- Some style violations; code requires effort to understand

Needs Improvement (0–8 pts)

- Poor or misleading names throughout
- Spaghetti code, deep nesting, or massive functions
- Significant style violations; very difficult to read

4. Design, Modularity & Algorithmic Choices (15 points)

Evaluates whether appropriate data structures, algorithms, and architectural decisions were made, and whether the student considered efficiency and modularity.

Excellent (13–15 pts)

- Chooses clearly appropriate and efficient data structures/algorithms for the problem
- Good modularity — code is broken into logical, reusable pieces
- Shows awareness of time/space complexity and justifies choices
- Extensible design (easy to add features or change requirements)

Good (10–12 pts)

- Mostly appropriate choices with minor sub-optimal decisions
- Reasonable modularity but some tight coupling or duplication
- Some consideration of efficiency, even if not optimal

Satisfactory (6–9 pts)

- Works but uses clearly inefficient or inappropriate approaches for the scale/problem
- Monolithic code with little modularity
- Little evidence of thinking about design or alternatives

Needs Improvement (0–5 pts)

- Inappropriate data structures or algorithms that make the solution unnecessarily complex or slow
- No meaningful modularity; everything in one function or class
- No apparent design thought process

5. Documentation, Comments & Reflective Explanation (10 points)

Assesses the quality of communication — both inline comments and any required written reflection/explanation.

Excellent (9–10 pts)

- Comments explain *why* decisions were made and any non-obvious logic
- Written explanation is clear, well-organized, and demonstrates reflection on learning
- No unnecessary comments; every comment adds value
- Student reflects on challenges faced and what they learned

Good (7–8 pts)

- Comments are mostly helpful but some are obvious or missing in complex areas
- Written explanation covers the main points adequately
- Some reflection present but could be deeper

Satisfactory (4–6 pts)

- Comments are sparse, obvious, or explain *what* instead of *why*
- Written explanation is brief, superficial, or mostly describes steps taken
- Limited evidence of reflection

Needs Improvement (0–3 pts)

- Little or no meaningful documentation or comments
- Written explanation missing, copied, or does not address the assignment requirements
- No visible reflection on the process or concepts

Important Notes for Teachers & LLM Use

Customization: Adapt criteria, point values, and specific indicators to your assignment. For example, add 'Must implement using recursion' or 'Must demonstrate understanding of hash table collision handling' as explicit expectations under the relevant criterion.

Combating AI-Generated Submissions: The Conceptual Understanding criterion + requirement for written explanation + process artifacts (git history, reflection) helps surface cases where students may not fully understand their own code.

Hybrid Evaluation: Use LLMs for speed, consistency, and detailed feedback. Always keep a human in the loop for final grades, especially on high-stakes work. Spot-check LLM scores periodically against your own judgment.

Condensed Version for LLM Prompts (copy-paste ready):

Evaluate the following student coding submission using this rubric (total 100 pts):

1. Functional Correctness & Robustness (30) — Excellent: works on all tests + edge cases + graceful errors. Good: mostly works, minor issues. Satisfactory: basic cases only, poor error handling. Needs Improvement: major bugs/crashes.

2. Conceptual Understanding (25) ★ — Excellent: clear accurate explanation of concepts + code choices reflect deep grasp + trade-offs discussed + strong "why" comments. Good: mostly accurate with minor gaps. Satisfactory: describes what more than why, some misconceptions. Needs Improvement: little understanding visible in code or explanation.

3. Code Quality (20) — Excellent: excellent naming, clean structure, follows idioms. Good: mostly good with minor issues. Satisfactory: vague names, large functions. Needs Improvement: poor names, unreadable.

4. Design & Algorithmic Choices (15) — Excellent: appropriate efficient structures, good modularity, justified choices. Good: mostly appropriate. Satisfactory: works but inefficient or monolithic. Needs Improvement: clearly wrong approach.

5. Documentation & Explanation (10) — Excellent: meaningful "why" comments + reflective written explanation. Good: adequate comments and explanation. Satisfactory: sparse/obvious comments, superficial explanation. Needs Improvement: missing or unhelpful.

For each criterion, give the score, quote 1-2 specific pieces of evidence from the student's code or explanation, and write 1-2 sentences of feedback. Output in clear sections. Flag anything surprising for human review.