PANS CODE RATING MODEL

Leave a Comment / By R&D TEAM / 25 January 2026

PANS Code Rating Model: A Universal Framework for Evaluating Code Quality (PCRM)

> *"Programs must be written for people to read, and only incidentally for machines to execute."*
> *— Harold Abelson*

Software lives far longer than its initial execution. Code is read, reviewed, modified, extended, and judged—often by people who were not present when it was first written. Yet most evaluations of code quality still stop at a shallow question: *Does it work?*

The **PANS Code Rating Model** is introduced to answer a deeper one: *How well does this code communicate, survive change, and behave under real-world conditions?* Just as PCEM provides a structured lens for understanding code, PANS provides a disciplined lens for **rating** it.

PANS is not about style preferences or personal opinions; it is about reducing ambiguity in how code quality is discussed, taught, reviewed, and improved. By grounding evaluation in clear, weighted principles, the model aims to create a shared standard for what good code truly means.

Introduction

The **PANS Code Rating Model** is a structured, principle-driven framework for evaluating code quality across all levels of complexity—from short scripts to large-scale, production-grade systems. Inspired by the need for a universal and repeatable standard, PANS provides a common language to assess *how well* code is written, not merely *whether* it works.

In practice, many codebases pass basic functional checks yet fail under real-world pressures such as scale, maintenance, collaboration, or change. The PANS model is designed to close this gap. It evaluates code along five critical dimensions that collectively determine long-term reliability, clarity, and adaptability. These dimensions are not tied to any specific language, framework, or paradigm; instead, they reflect fundamental engineering principles that remain valid over time.

At its core, the model distinguishes between **code that runs** and **code that endures**. It emphasizes minimizing the distance between *intent*, *implementation*, and *understanding*—for individual developers, teams, reviewers, educators, and even AI systems analyzing or generating code.

The five pillars of the PANS Code Rating Model are:

- **P – Program Structure**: How the code is organized, modularized, and architected.
- **A – Accuracy of Functionality**: Whether the code truly and logically solves the intended problem.
- **N – Naming and Readability**: How clearly the code communicates its purpose to humans.
- **S – Safety and Error Handling**: How well the code anticipates and manages failures and edge cases.
- **S – Scalability and Reusability**: How effectively the code supports growth, change, and extension without breakage.

Together, these dimensions form a **quantitative and qualitative rating lens**. Each pillar in the PANS model is assigned a **weighted score**, allowing reviewers to compute an overall code quality score while still preserving nuanced judgment. This makes PANS suitable not only for discussion and explanation, but also for comparison, benchmarking, and progressive improvement.

Weighted Scoring Overview

The PANS Code Rating Model assigns a total of **100 points**, distributed across the five pillars based on their impact on long-term code quality:

- **P – Program Structure (25 points)**
  Measures architectural clarity, modularity, reuse, avoidance of redundancy, and disciplined organization.
- **A – Accuracy of Functionality (25 points)**
  Evaluates whether the code correctly, logically, and completely solves the intended problem without unintended behavior.
- **N – Naming and Readability (15 points)**
  Assesses how clearly intent is communicated through naming, structure, and human-oriented readability.
- **S – Safety and Error Handling (15 points)**
  Rates robustness against failures, edge cases, invalid states, and operational uncertainties.
- **S – Scalability and Reusability (20 points)**
  Determines readiness for growth, change, and extension without breaking existing behavior.

The final PANS score enables objective comparison between codebases or revisions, while the individual pillar scores highlight *where* and *why* quality gaps exist. Subsequent sections define each pillar, its evaluation criteria, and how partial scoring should be applied in real-world reviews.

P – Program Structure (25 Points)

Definition

Program Structure evaluates how code is organized, decomposed, and architected. It examines whether the codebase reflects clear separation of concerns, logical modularity, disciplined reuse, and sound structural decisions that support understanding and change over time.

This pillar is intentionally weighted heavily because structure determines how easily code can evolve. Poorly structured code may function correctly today, yet become fragile, opaque, and expensive to maintain as requirements grow.

What This Pillar Measures

Program Structure focuses on *how the solution is constructed*, independent of what it computes. Key aspects include:

- Logical decomposition into modules, classes, or functions
- Appropriate use of abstractions and libraries
- Elimination of unnecessary duplication (DRY principle)
- Avoidance of hard-coded values, secrets, and configuration leaks
- Consistent and meaningful data structure usage
- Clear boundaries between layers (e.g., data, business logic, interfaces)

Well-structured code allows a reader to infer intent from layout alone, before reading individual lines.

Scoring Criteria

22–25 (Excellent Structure)

- Code is cleanly modularized with clear responsibilities
- No unnecessary declarations or unused components
- Reusable libraries and utilities are leveraged appropriately
- Configuration, secrets, and environment-specific values are externalized
- Data structures are chosen intentionally and used consistently

- Changes can be localized without ripple effects

15–21 (Adequate Structure)

- Overall organization is reasonable but uneven
- Some duplication or minor over-structuring exists
- Partial hard-coding or mixed responsibilities appear
- Refactoring is possible but not trivial

7–14 (Weak Structure)

- Logic is tightly coupled and poorly decomposed
- Repeated code blocks exist across the codebase
- Overuse or misuse of global state or shared mutable data
- Structural changes risk breaking unrelated functionality

0–6 (Poor Structure)

- Monolithic or tangled logic with no clear boundaries
- Extensive hard-coded values, credentials, or constants
- Copy-paste–driven development
- Code cannot be safely modified without deep rework

Common Structural Anti-Patterns

- “God” classes or functions doing unrelated work
- Repetition of logic instead of abstraction
- Configuration mixed directly into business logic
- Premature or unnecessary abstraction without clear benefit
- Ignoring native language or framework conventions

Example: Poor vs Well-Structured Code

Scenario: Processing user registration data and saving it to a database.

Poorly Structured Approach

- Validation, business rules, database access, and logging are written inside a single function
- Configuration values (timeouts, table names) are hard-coded
- Logic is duplicated across multiple registration flows

```
registerUser(data):
  if data.email contains '@' then
    connect to DB at "localhost:3306"
    insert user
    print "user saved"
  else
    print "invalid email"
```

Even if this code works, it scores low (≈6–8/25) because:

- Responsibilities are mixed
- Hard-coded configuration exists
- Logic is not reusable or extensible

Well-Structured Approach

- Validation is isolated
- Business logic is separated from persistence
- Configuration is externalized

- Each unit has a single, clear responsibility

```
validateUser(data)
createUser(data)
saveUser(user, repository)
```

Here, structure alone communicates intent. Changes such as adding a new validation rule or switching databases affect only one module. This version scores high (≈22–25/25), even before evaluating correctness or readability.

Key Insight

Strong program structure minimizes **structural entropy**. It ensures that as features increase, complexity grows linearly rather than exponentially. In the PANS model, no amount of functional correctness can compensate for structurally unsound code beyond a capped score in this pillar.

The next section evaluates whether the structured code actually *solves the right problem correctly*—the **Accuracy of Functionality** pillar.

A – Accuracy of Functionality (25 Points)

Definition

Accuracy of Functionality measures whether the code correctly, completely, and logically fulfills its intended purpose. This pillar evaluates *what the code does*, not how elegant or readable it appears. A solution that executes without errors but produces incorrect, incomplete, or misleading results scores poorly under this dimension.

This pillar draws a critical distinction between **"code that runs"** and **"code that is right."** In the PANS model, functional accuracy is non-negotiable: no amount of structural cleanliness or readability can compensate for incorrect behavior.

What This Pillar Measures

Accuracy of Functionality focuses on the alignment between **problem intent** and **program behavior**. Key aspects include:

- Correct handling of core use cases and business rules
- Logical soundness of the chosen approach
- Termination guarantees (no infinite loops or runaway recursion)
- Absence of unintended side effects
- Deterministic and predictable outcomes
- Fitness of the solution (right algorithm for the right problem)

The evaluation asks not only *"Does this produce output?"* but *"Does this output represent the correct solution under all expected conditions?"*

Scoring Criteria

23–25 (Correct and Fit-for-Purpose)

- All defined requirements are met accurately
- Logic is sound across normal and edge scenarios
- Algorithmic approach is appropriate to the problem domain
- No infinite loops, deadlocks, or unintended state mutations
- Behavior matches stated or implied intent exactly

16–22 (Mostly Correct)

- Core functionality works as intended
- Minor logical gaps or unhandled scenarios exist
- Some assumptions are implicit rather than enforced
- Errors are rare but possible under boundary conditions

8–15 (Partially Correct)

- Code works for common cases only
- Edge cases are ignored or mishandled
- Logical shortcuts lead to incorrect results in non-trivial scenarios
- Solution “appears” to work but lacks rigor

0–7 (Incorrect or Misleading)

- Code produces wrong or inconsistent results
- Intended problem is misunderstood or incompletely solved
- Infinite loops, incorrect termination, or invalid state transitions occur
- Output cannot be trusted even if execution succeeds

Common Functional Anti-Patterns

- Confusing example inputs with general solutions
- Hard-coding logic that only works for known cases
- Ignoring boundary conditions (empty input, limits, overflow)
- Using brute-force solutions where correctness silently degrades
- Treating absence of runtime errors as proof of correctness

Example: “It Works” vs “It Is Correct”

Scenario: Polling a background job until completion.

Superficially Working Code

```
while job.status != "COMPLETED":
  fetchLatestStatus(job)
```

This code executes and may work during testing, but it has a critical flaw: there is no termination guard. If the job enters a failed, stuck, or unexpected state, the loop runs indefinitely, consuming resources and potentially blocking the system. Despite compiling and running, the solution is logically unsafe and incomplete. This scores low (≈5–8/25).

Functionally Accurate Code

```
attempts = 0
while attempts < MAX_POLLS:
  status = fetchLatestStatus(job)
  if status == "COMPLETED":
    markSuccess()
    return
  if status == "FAILED":
    handleFailure()
    return
  attempts = attempts + 1

handleTimeout()
```

Here, all execution paths are explicitly defined. The loop has termination guarantees, failure states are handled, and system behavior remains deterministic even under abnormal conditions. The logic fully aligns with the intent of safe job monitoring. This version scores high (≈23–25/25).

Key Insight

Functional accuracy is the **foundation** of code quality. In the PANS model, structural elegance and readability are evaluated only *after* correctness is established. Code that merely appears to work under limited or ideal conditions is treated as fundamentally incomplete.

The next section examines how clearly this correct behavior is communicated to humans—the **Naming and Readability** pillar.

N – Naming and Readability (15 Points)

Definition

Naming and Readability evaluate how effectively code communicates intent to a human reader. This pillar answers a simple but critical question: *Can someone—possibly a novice, a replacement , a functional person like a BA or PO or even the original author after years—understand what this code is doing without external explanation?*

Unlike functional accuracy, this pillar does not measure *correctness*. Instead, it measures **cognitive clarity**. Code is read far more often than it is written, and poor readability increases maintenance cost, refactoring risk, onboarding time, and defect probability.

What This Pillar Measures

Naming and Readability focus on how clearly intent is expressed through:

- Descriptive variable, function, and class names
- Names that reveal *why* something exists, not just *what* it stores
- Logical flow that can be followed top-to-bottom
- Minimal mental translation required to understand behavior
- Consistency in naming conventions and abstractions

Well-written code should be understandable even to a **novice programmer**, provided they understand the basic syntax of the language.

Scoring Criteria

13–15 (Highly Readable)

- Variable and function names clearly express intent
- Code reads almost like natural language
- A new reader can explain the logic after a single pass
- Minimal comments are needed because names carry meaning

9–12 (Readable with Effort)

- Names are generally reasonable but occasionally vague
- Reader must infer intent from surrounding logic
- Understanding requires re-reading sections

4–8 (Poor Readability)

- Abbreviated, generic, or misleading names dominate
- Reader must mentally simulate code to understand it
- High cognitive load even for simple logic

0–3 (Unreadable)

- Single-letter or meaningless identifiers
- Intent is completely obscured
- Code cannot be safely modified due to lack of clarity

Common Readability Anti-Patterns

- Using short or cryptic variable names without necessity
- Naming based on implementation rather than intent
- Overloading the same name for different purposes
- Writing code that requires comments to explain basic logic
- Optimizing for brevity instead of clarity

Example: Unfriendly vs Friendly Code

Scenario: Calculating whether a user is eligible for a discount.

Hard to Understand (Technically Correct)

```
if a > 60 and b == 1:
  c = d * 0.9
```

A novice reader cannot easily determine:

- What `a`, `b`, `c`, or `d` represent
- Why the condition exists
- What the calculation means

Even if functionally correct, this scores low (≈4–6/15) due to poor intent communication.

Clear and Novice-Friendly

```
if userAge > 60 and isPremiumMember:
  discountedPrice = originalPrice * 0.9
```

Here, intent is immediately obvious—even to someone new to the codebase or language. The logic can be explained without running the program. This scores high (≈13–15/15).

Key Insight

Readable code reduces the **time-to-understanding**. In the PANS model, readability measures how well intent survives across time, team changes, and experience levels. Code that only experts can decipher is treated as fragile, regardless of correctness.

The next section evaluates how safely the code behaves under failure and uncertainty—the **Safety and Error Handling** pillar.

S – Safety and Error Handling (15 Points)

Definition

Safety and Error Handling measure how well code anticipates, detects, and responds to failure conditions. This pillar evaluates whether the code is written only for the *happy path* or whether it remains correct, predictable, and recoverable when things go wrong.

In real-world systems, failure is not exceptional—it is expected. Networks fail, inputs are invalid, dependencies timeout, and resources become unavailable. Safe code explicitly plans for these realities rather than assuming ideal conditions.

What This Pillar Measures

Safety and Error Handling focus on the code's defensive behavior under uncertainty, including:

- Handling of null, empty, zero, or invalid inputs
- Clear detection and reporting of errors
- Retry, timeout, and fallback strategies
- Rollback or compensation on partial failure
- Prevention of silent failures or corrupted state
- Predictable behavior during exceptional conditions

This pillar does **not** evaluate whether errors exist, but whether they are **managed deliberately and safely**.

Scoring Criteria

13–15 (Robust and Defensive)

- All failure paths are explicitly handled
- Errors are detected early and surfaced clearly
- Retries, timeouts, or fallbacks are defined where applicable
- System state remains consistent after failure
- No silent or ambiguous outcomes

9–12 (Moderately Safe)

- Common failure cases are handled
- Some edge failures may propagate unchecked
- Error handling exists but is inconsistent or incomplete

4–8 (Fragile)

- Code assumes valid input and available dependencies
- Failures cause crashes, undefined behavior, or corrupted state
- Error handling is reactive rather than designed

0–3 (Unsafe)

- No explicit error handling
- Failures are ignored or swallowed silently
- System behavior under error is unpredictable

Common Safety Anti-Patterns

- Writing code only for the happy path
- Catching errors without meaningful handling
- Ignoring return values or error codes
- Assuming external systems are always available
- Leaving partial updates without rollback

Example: Happy Path vs Safe Code

Scenario: Fetching user data from an external service.

Happy-Path-Only Code

```
user = fetchUserFromService(userId)
processUser(user)
```

This code assumes:

- The service is reachable
- The user exists
- The response is valid

If any assumption fails, the system may crash or behave unpredictably. Despite being simple and readable, this scores low (≈4–6/15).

Safe and Defensive Code

```
try:
  user = fetchUserFromService(userId, timeout=3s)
  if user is null:
    handleUserNotFound()
    return
  processUser(user)
except TimeoutError:
  retryOrFallback()
except ServiceError as e:
  logError(e)
  handleServiceFailure()
```

Here, failure modes are explicit. Timeouts are bounded, invalid states are handled, and system behavior remains controlled even when dependencies fail. This scores high (≈13–15/15).

Key Insight

Safe code assumes failure by default. In the PANS model, handling only the happy path is treated as incomplete engineering. Robust error handling ensures that correctness survives real-world uncertainty.

The next section evaluates whether this safe code can grow and adapt over time—the **Scalability and Reusability** pillar.

S – Scalability and Reusability (20 Points)

Definition

Scalability and Reusability assess whether code is designed to handle growth—both in usage and in functionality—without requiring disruptive rewrites. This pillar differentiates *working code* from *production-ready code*.

Scalability refers not only to performance under increased load, but also to **design scalability**: how well the codebase absorbs new requirements. Reusability evaluates whether existing components can be safely extended or reused across contexts without duplication or modification.

What This Pillar Measures

This pillar focuses on the future-facing qualities of code, including:

- Ability to handle increased users, data volume, or execution frequency
- Ease of adding new features without modifying existing logic
- Adherence to the Open–Closed Principle (open for extension, closed for modification)
- Absence of tight coupling that blocks reuse
- Configurability instead of hard-coded behavior

- Predictable performance characteristics as scale increases

The evaluation asks: *If requirements double or change tomorrow, does this code bend—or break?*

Scoring Criteria

18–20 (Production-Ready)

- Code supports growth in load and functionality with minimal change
- New behavior can be added via extension, not modification
- Components are reusable across workflows or services
- No hidden assumptions about scale or usage

13–17 (Scalable with Constraints)

- Code handles moderate growth but has known limits
- Some refactoring is required for major extensions
- Reuse is possible but not seamless

7–12 (Limited Scalability)

- Code works only within current constraints
- New features require modifying existing logic
- Performance or complexity degrades quickly with growth

0–6 (Non-Scalable / Disposable)

- Logic is tightly coupled and rigid
- Growth requires rewriting core components
- Code is effectively single-use

Common Scalability Anti-Patterns

- Hard-coding limits, thresholds, or behaviors
- Designing for current usage only
- Mixing feature logic instead of extending it
- Copy-pasting code to support new use cases
- Assuming load or complexity will remain static

Example: Working Code vs Scalable Code

Scenario: Sending notifications to users.

Working but Non-Scalable Code

```
sendEmail(user)
```

When a new requirement arrives (SMS, push notifications), the existing function must be modified repeatedly:

```
if channel == "email":
  sendEmail(user)
else if channel == "sms":
  sendSMS(user)
```

Each new channel increases complexity and risk. This approach scores low (≈6–8/20).

Scalable and Reusable Code

```
interface NotificationChannel:
  send(user)

sendNotification(channel: NotificationChannel, user)
```

New notification types are added by implementing the interface, not by changing existing logic. Growth is linear, behavior is isolated, and reuse is natural. This scores high (≈18–20/20).

Key Insight

Scalable code anticipates change. In the PANS model, the highest scores are reserved for designs where growth—whether in users, features, or load—does not increase fragility. Code that must be rewritten to evolve is treated as incomplete engineering, even if it works today.

PANS Code Rating Summary Table

The table below provides a consolidated view of the PANS Code Rating Model, enabling quick evaluation and comparison during reviews.

Pillar	Weight	What It Evaluates	Low-Score Indicators	High-Score Indicators
P – Program Structure	25	Architectural organization, modularity, reuse, avoidance of hard-coding	Monolithic logic, duplication, hard-coded configs, tangled responsibilities	Clear separation of concerns, modular design, reusable components
A – Accuracy of Functionality	25	Correctness, logical soundness, solution fitness, termination guarantees	Infinite loops, incorrect logic, partial solutions, silent failures	Correct behavior across normal and edge cases, deterministic outcomes
N – Naming & Readability	15	Human understandability, intent clarity, cognitive load	Cryptic names, excessive comments, hard-to-follow logic	Self-explanatory names, clear flow, novice-friendly code
S – Safety & Error Handling	15	Defensive behavior under failure, error management	Happy-path-only logic, ignored errors, unpredictable failure behavior	Explicit error handling, retries, timeouts, safe recovery
S – Scalability & Reusability	20	Ability to grow in users, load, and features without breakage	Tight coupling, modification-driven growth, rigid limits	Extension-based growth, reusable abstractions, scale readiness

Total Score: 100 Points

This table is intended as a quick-reference guide. Detailed scoring should always be derived from the full pillar definitions and examples provided in the sections above.

Worked Example: Applying the PANS Code Rating Model

Scenario: A simple service that processes orders and sends a confirmation.

```
processOrder(order):
  if order.items.length == 0:
    return

  total = 0
  for i in order.items:
    total = total + i.price

  sendEmail(order.userEmail, total)
```

Step-by-Step PANS Evaluation

P – Program Structure (25)

- Single function mixes validation, calculation, and notification
- No modular separation
  Score: 12 / 25

A – Accuracy of Functionality (25)

- Core logic works for basic cases
- Missing handling for invalid prices, duplicate items, or failures in email sending
- Silent return on empty order is ambiguous
  Score: 17 / 25

N – Naming & Readability (15)

- Names are understandable but generic
- Intent is mostly clear even to a novice
  Score: 11 / 15

S – Safety & Error Handling (15)

- No error handling for email failure or invalid input
- Happy-path-only implementation
  Score: 5 / 15

S – Scalability & Reusability (20)

- Logic is tightly coupled
- Adding tax, discounts, or new notification channels requires modifying the function
  Score: 8 / 20

Final PANS Score

Total: 53 / 100
The code works, but it is not production-ready. The score clearly shows where improvement is required and why.

Conclusion

The PANS Code Rating Model provides a structured, repeatable way to reason about code quality beyond subjective opinion. By separating concerns into five weighted pillars, PANS enables engineers, reviewers, educators, and AI systems to evaluate code with clarity, consistency, and intent.

Rather than asking whether code merely runs, PANS asks whether it is **correct, understandable, safe, and capable of growth**. This shift in evaluation mindset is essential for building software that endures real-world complexity.

When applied consistently, the model reduces ambiguity in reviews, improves learning outcomes, and establishes a shared standard for what *good code* truly means.

Author's Note and Usage Rights

The **PANS Code Rating Model** is a conceptual and educational framework, similar in spirit to fundamental models used in mathematics, engineering, and physics. It is **standardized way of thinking about code quality**. The author explicitly grants permission to use this model for training, evaluating, and reasoning within Large Language Models (LLMs), as well as for ai building tooling, frameworks, or methodologies that aim to explain, analyze, or teach code

You are free to:
Use the PANS model in code reviews, interviews, teaching, documentation, and research
Reference or quote the model with attribution
Apply the scoring system to personal, academic, or commercial projects

You may not:
Claim authorship of the model
Misrepresent modified versions as the original PANS model
If you adapt or extend the framework, clearly state that it is a derivative interpretation.
The intent of PANS—like PCEM—is to improve clarity, raise engineering standards, create a common language for understanding code across experience levels, domains, and tools.

← Previous Post

Leave a Comment

Your email address will not be published. Required fields are marked *

Type here..

Name*

Email*

Website

Save my name, email, and website in this browser for the next time I comment.

Post Comment