PANS CODE EXPLANATION MODEL

"While we may come from different places and speak in different tongues, our hearts beat as one."
- Albus Dumbledore in Harry Potter and the Goblet of Fire

Coding is like Magic, we create an entire software or app starting from scratch and create something which feels alive, reasonable and functional out of thin air. Programmers across the world write code in different languages, follow different paradigms, and work within vastly different problem spaces. Some build simple scripts to automate daily tasks; others engineer systems that power industries and impact millions. The syntax may change, the tools may evolve, and the contexts may differ—but at the core, the intent remains the same: to solve problems with clarity and purpose.

Stop thinking code as just few lines which you compile or need to merge into the main branch. Code, much like language, is deeply expressive.

But its true meaning is often locked behind layers of abstraction, convention, and experience. A line of code that feels obvious to one engineer may be opaque to another. This gap is not a failure of intelligence or skill; it is a failure of shared understanding. This is where a universal approach to code understanding becomes essential. Not as a tool that inspects or evaluates, but as a common framework that allows any piece of code to be explained clearly, consistently, and meaningfully—regardless of who wrote it or who is studying it.

Introduction

This paper introduces the **PANS Code Explanation Model (PCEM)**, a foundational framework for code interpretation, positioned analogously to a mathematical theorem or a principle in fundamental physics. Rather than functioning as an analytics or inspection mechanism, the model defines a standard, structured way through which any snippet of code—ranging from elementary scripts to highly complex, industry-grade algorithms—can be explained with clarity and consistency. The model abstracts code into essential conceptual dimensions that reveal intent, logical structure, and practical application, independent of programming language or implementation detail.

The formulation of this model is grounded in the author's fifteen years of experience in product development, encompassing the design and delivery of large-scale, production-grade systems. Its structure reflects repeated patterns observed while collaborating with engineers who solved real-world problems under constraints of performance, reliability, and maintainability. Through these experiences, the model distils practical reasoning processes into a formalized framework that can be applied universally.

The following seven steps represent the **PANS Code Explanation Model (PCEM)**. They are designed to be applied sequentially, ensuring that understanding is not accidental, but intentional and complete.

Step 1 : Understand Intent

"Define the problem"

Every piece of code exists for a reason. Before examining syntax, logic, or structure, true understanding begins by identifying intent. Ask a simple fundamental question "what is it doing" or more precisely **"what problem is it solving".**
Intent defines the boundary within which all implementation decisions make sense. Without it, even well-written code appears arbitrary, and complexity becomes indistinguishable from confusion. By

anchoring understanding to the underlying problem—whether functional, business-driven, or technical—this step establishes context and purpose.

Example 1 : Consider a short function that iterates over a list of numbers and returns the largest value. At the intent level, the goal is not “looping through an array” or “using a comparison operator.” The intent is to identify the maximum value from a given set of inputs. Without establishing this intent first, the code risks being read as a sequence of mechanical steps rather than a solution to a defined problem.

Example 2 : Now consider a large, distributed service responsible for processing millions of financial transactions per day, involving validation, fraud detection, reconciliation, and audit logging.
At the intent level, the problem is not microservices, message queues, or concurrency control. The core intent is to safely, accurately, and traceably process financial transactions at scale while meeting regulatory and performance constraints. Only when this intent is understood do architectural choices—such as idempotency mechanisms, event-driven workflows, or eventual consistency—become meaningful rather than overwhelming.

In both cases, the scale differs dramatically, but the principle remains constant: intent is the anchor. Whether interpreting a single function or an enterprise-grade system, starting with what problem is being solved transforms code from lines of syntax into purposeful design.

Step 2 Define Inputs and Outputs

“Define the transaction”

Once the intent is understood, the next step is to clearly identify what the code takes in and what it gives back. At its core, every piece of code can be viewed as a transaction : you give something you get back something.

This step establishes a simple but powerful contract. It answers the question, **if I provide this code with certain inputs, what outputs should I expect?** Without this clarity, understanding becomes fragile and dependent on reading implementation details too early.

Inputs are all information the code is to be given to start its job. This includes parameters, user input, or any external data the code reads. If a value influences the result, it is an input—even if it is not explicitly passed as an argument.

Outputs are the results produced by the code. These may include return values, responses sent to a client, values displayed on a screen, or data passed to another part of the system. Outputs define what the code promises to deliver when it finishes executing.

In PANS Code Explanation Model, this step deliberately avoids how the result is produced. By focusing only on inputs and outputs, the code is understood as a clear transaction rather than a sequence of instructions.

Example 1: Consider a function that takes two numbers and returns their sum.
Inputs: Two numeric values.
Outputs: A single numeric value representing their sum.

Example 2: Consider an API endpoint that generates a monthly account statement.
Inputs: User identifier, date range, and account data retrieved from upstream systems.
Outputs: A structured statement response, such as a PDF or JSON payload.

At this level, loops, operators, and syntax are irrelevant. The code is understood purely in terms of what it consumes and what it produces. Even though the internal logic may be extensive, understanding the inputs and outputs provides a stable foundation. Any deeper analysis can now build on this contract with confidence.

This step answers a simple but critical question: when this code runs and **what does this code needs to start and what is the end goal of this**.

Step 3: Identify What the Code Changes

"Understand the impact"

After defining what goes into the code and what comes out, the next step is to understand what the code *changes* while producing that result. Not all effects of code are visible in its output. Many changes happen quietly in the background and can significantly influence system current state or behaviour.

This step answers a critical question**: what is different in the system after this code runs?** These changes may persist beyond execution and affect future behaviour, making them essential to understanding the code fully.

Changes can take many forms. The code may update stored data, modify shared variables, write to files, send information over a network, or record logs. In user-facing applications, it may also change what the user sees on the screen. Identifying these changes prevents surprises and helps explain why running the same code twice may not always produce the same outcome.

In PANS Code Explanation Model, this step separates results from impact. While outputs describe what the code returns immediately, changes describe how the surrounding system is altered as a consequence of executing the code.

Example 1: Consider a function that increments a global counter each time it is called.
What it changes: The value of the counter stored in memory.
Even if the function returns nothing, it still has an effect. Future calls behave differently because the underlying value of the global counter has changed.

Example 2: Consider a service that processes a user profile update.
What it changes: User records in the database, cached profile data, audit logs, and possibly downstream systems that rely on profile information.
Although the API response may simply confirm success, the real impact lies in the changes made across the system. Understanding these changes is essential for reasoning about data consistency, debugging issues, and predicting downstream effects.

By explicitly identifying what the code changes, this step ensures that understanding goes beyond surface-level behaviour and captures the lasting impact of execution.

Step 4: Identify Dependencies and Why are they needed

"The prerequisites "

Once it is clear what the code is meant to do, what goes in and comes out, and what it changes, the next step is to understand what the code *depends on* to function correctly. No meaningful code operates alone; it relies on other pieces of code, libraries, systems, or assumptions.

This step answers two related questions: **what does this code rely on, and why does it rely on it?** Identifying dependencies without understanding their purpose leads to shallow comprehension. Understanding why a dependency exists reveals design intent and trade-offs.

Dependencies can include language features, helper functions, internal modules, third-party libraries, frameworks, configuration files, or external services. Some dependencies exist to simplify logic, some to improve performance, and others to enforce consistency or security. Each dependency represents a decision made by the author.

In PANS Code Explanation Model, dependencies are not treated as black boxes. Instead, they are examined in terms of the role they play. A dependency should justify its presence by enabling behaviour that would otherwise be harder, riskier, or less maintainable to implement.

Example 1: Simple Code
Consider a function that formats a date using a standard library utility.
Dependencies: A built-in date formatting library.
Why it is needed: To handle locale-specific formatting correctly and avoid manual string manipulation.
The dependency reduces complexity and prevents common errors, making the code easier to trust and maintain.

Example 2: Complex System
Consider a payment processing service.
Dependencies: A payment gateway SDK, encryption libraries, configuration services, and a logging framework.
Why they are needed: The gateway SDK handles secure transaction processing, encryption libraries protect sensitive data, configuration services allow safe runtime changes, and logging supports monitoring and compliance.

By identifying dependencies and their purpose, the code becomes understandable as a set of deliberate choices rather than an arbitrary collection of imports.
This step ensures that readers not only know *what* the code uses, but why it uses it—laying the groundwork for evaluating reliability, risk, and future change.

Step 5: Break the Code into Core Modules
"The Building Blocks"

After identifying dependencies, the next step is to understand how the code is internally organized. Most non-trivial code is not a single block of logic; it is composed of smaller parts, each responsible for a specific role. This step focuses on breaking the code down into those core modules.

This step answers the question: **how is the responsibility of this code divided?** By identifying distinct modules, functions, or logical sections, the overall behaviour becomes easier to reason about. Each module can be understood independently before being composed into the full system.

Core modules may take different forms depending on scale. In simple programs, they may be individual functions or classes. In larger systems, they may be services, layers, or components. What matters is not the syntax, but the separation of concerns—what each part is responsible for and what it deliberately does not handle.

In the PANS Code Explanation Model, breaking code into modules is not about refactoring or design critique. It is about creating a mental map. Once this map exists, complexity becomes manageable, and reasoning shifts from "reading code" to "understanding roles."

Example: Single-Page Backend Handler
Consider a single-page backend function that handles an HTTP request to place an order. The code spans one file and includes validation, business logic, and external interactions.
At first glance, the code appears dense and difficult to follow. Applying this step reveals a clear structure:

1. **Input Validation Module**
   Responsible for checking request parameters, ensuring required fields are present, and rejecting invalid data early.
2. **Authorization Module**
   Verifies user identity and permissions before allowing the operation to proceed.
3. **Business Logic Module**
   Applies pricing rules, calculates totals, applies discounts, and determines order eligibility.
4. **Persistence Module**
   Saves the order to the database and updates related records such as inventory.
5. **Integration Module**
   Communicates with external services such as payment gateways or notification systems.
6. **Response Construction Module**
   Builds and returns the final HTTP response.

Although all of this logic exists in a single page of code, breaking it into these core modules transforms understanding. Instead of tracking dozens of variables and conditionals, the reader reasons about six responsibilities executed in sequence.
This step allows complex code to be understood at a glance. Once the modules are identified, deeper analysis becomes targeted and intentional rather than overwhelming.

Step 6: Unlock the Logic

"Understand the how"

After breaking the code into core modules, the final barrier to understanding is removed by following the logic as it actually flows. At this stage, the structure is already clear, and the pieces are in place. What remains is to see **how those pieces work together.**

This step focuses on the execution path—the order in which modules run, the conditions that guide decisions, and the transformations that occur along the way. With inputs, outputs, effects, dependencies, and responsibilities already defined, the code no longer feels unpredictable.

What once appeared as magic or an incantation now reveals itself as **method and mechanism**. Each condition has a reason, each loop has a purpose, and each function call fits into a known role. The logic becomes traceable rather than mysterious, and the code stops feeling alien.

In PANS Code Explanation Model, this step represents the transition from familiarity to mastery. The reader is no longer observing the code from the outside, but reasoning through it confidently, step by step. The "how" is no longer hidden—it is unlocked.

Example : Consider a function that checks whether a user is eligible for a discount.
Once the modules are identified—input reading, rule checking, and result generation—the logical flow is straightforward:

1. The user’s data is read.
2. Eligibility conditions are evaluated.
3. A decision is made and returned.

There is no spell to decipher. The logic follows a clear path, governed by rules that are now visible and understood.

Step 7: Understand the Limits

“Know where the code holds—and where it does not”

No code is absolute. Just as every physical structure has load limits and every scientific model has assumptions, every piece of software operates within defined boundaries. Understanding these limits is the final step toward complete code comprehension.

This step asks critical questions: **How resilient is the code? Under what conditions does it fail? What assumptions does it make about its inputs, environment, or dependencies?** Limits reveal the edges of correctness, performance, and reliability.

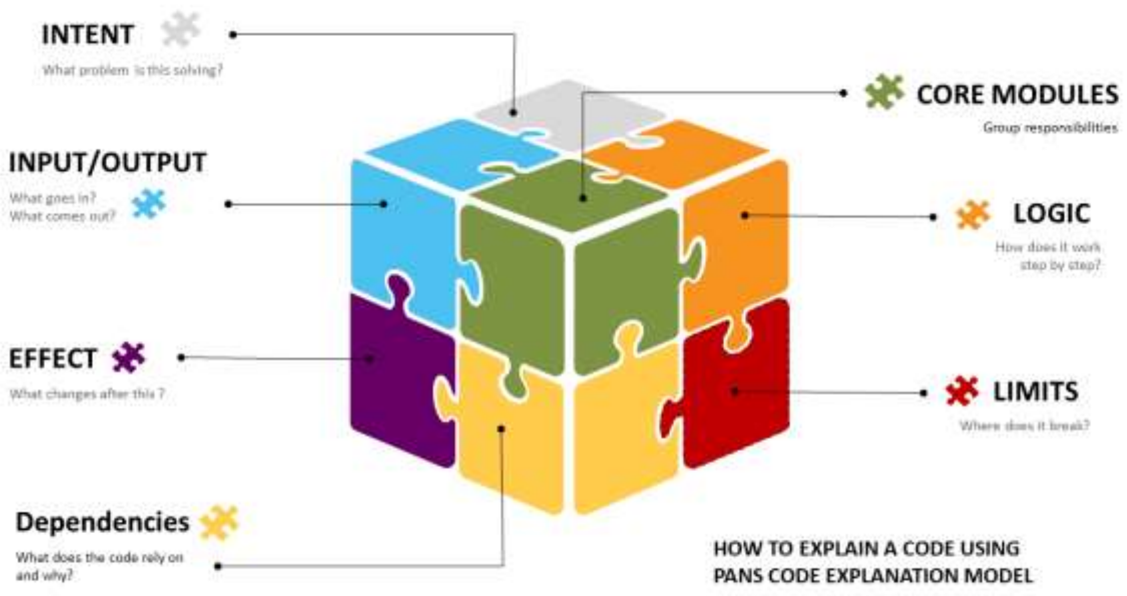
Limits appear in many forms. They include edge cases the code may not handle gracefully, constraints imposed by time, memory, or external systems, and the quality of error handling when things go wrong. A piece of code may work perfectly under normal conditions yet behave unpredictably when pushed beyond its design envelope.

An effective analogy is that of a bridge. A bridge is not weak because it has a maximum load; it is safe because that limit is known and respected. Similarly, robust code is not defined by the absence of failure, but by how clearly its boundaries are understood and how gracefully it responds when those boundaries are crossed.

In PANS Code Explanation Model, this step completes the transition from “working code” to *trusted code*. By identifying limits, engineers can judge suitability, anticipate failure modes, and make informed decisions about reuse, extension, or replacement.

Understanding what code cannot do is just as important as understanding what it can. Only then is comprehension complete.

To reinforce this approach, the model is accompanied below by a cube puzzle–based pictorial representation. Each face of the cube represents a distinct dimension of code understanding—intent, inputs and outputs, effects, dependencies, structure, logic, and limits. Individually, no single face reveals the full picture. Only when all sides are examined and aligned does the code become truly understandable. This visualization reflects how real-world code comprehension works in practice: understanding emerges not from a single viewpoint, but from assembling multiple perspectives into a coherent whole.


INTENT
What problem is this solving?
INPUT/OUTPUT
What goes in?
What comes out?
EFFECT
What changes after this ?
Dependencies
What does the code rely on and why?
CORE MODULES
Group responsibilities
LOGIC
How does it work step by step?
LIMITS
Where does it break?
HOW TO EXPLAIN A CODE USING PANS CODE EXPLANATION MODEL

Conclusion

Code understanding is often treated as an implicit skill—something acquired through experience rather than articulated through structure. As software systems grow in scale and complexity, this assumption no longer holds. Teams are distributed, codebases outlive their original authors, and decisions embedded in code must be understood, questioned, and evolved over time.

The PANS Code Explanation Model provides a structured yet flexible framework to address this challenge. By progressing from intent to limits, the model enables engineers, reviewers, and learners to reason about code in a deliberate and repeatable way. It scales naturally from simple scripts to complex, production-grade systems and applies equally well to debugging, code reviews, onboarding, system design discussions, and architectural decision-making.

In real-world scenarios, the model helps demystify unfamiliar code, reduce misinterpretation during collaboration, and surface hidden assumptions and risks early. For complex problem domains—such as distributed systems, financial platforms, or safety-critical software—it provides a common language to align understanding across technical and non-technical stakeholders.

Ultimately, the PANS Code Explanation Model reframes code not as an opaque artefact to be deciphered, but as a system of intent, decisions, and constraints that can be understood, explained, and trusted.

Author’s Note on Usage and Rights

The PANS Code Explanation Model is made freely available for use by the author. It may be used without restriction for academic, educational, and research purposes, including but not limited to classroom teaching, curriculum design, technical training, code reviews, and software engineering education.