# Harness: Structured Pre-Configuration for Enhancing LLM Code Agent Output Quality

*A Controlled Experiment with 15 Software Engineering Tasks*

**Minho Hwang**

revfactory@gmail.com

March 5, 2026

**Abstract**

Large Language Model (LLM)-based code agents such as Claude Code have demonstrated remarkable capability in generating functional software. However, the quality of their output—particularly in architectural design, test coverage, and extensibility—varies significantly depending on structural guidance provided. This paper introduces **Harness**, a systematic pre-configuration framework that enhances LLM code agent output through structured project scaffolding. We conduct a controlled A/B experiment across 15 software engineering tasks spanning three difficulty levels (Basic, Advanced, Expert) and evaluate outputs across 10 quality dimensions. Results demonstrate that Harness improves average quality from 49.5 to 79.3 points (out of 100), a **60% improvement**. Critically, we find a strong positive correlation between task difficulty and effectiveness: Basic tasks show +23.8 improvement, Advanced +29.6, and Expert +36.2—a **52% increase in delta** from Basic to Expert. These findings suggest that structured pre-configuration becomes increasingly essential as tasks grow in complexity, and that the primary bottleneck for LLM code agents is not functional capability but structural organization.

**Keywords:** LLM Code Agents, Software Engineering, Code Quality, AI-Assisted Development, Prompt Engineering, Structured Generation

## 1. Introduction

The emergence of LLM-based code agents has transformed software engineering workflows. Tools such as Claude Code [1], GitHub Copilot Workspace [2], and Cursor [3] can generate complete software projects from natural language descriptions. While these tools excel at producing functional code, practitioners have observed inconsistencies in non-functional quality attributes—architecture, testing, documentation, and extensibility [4].

In practice, this means that while an LLM code agent can quickly produce a working prototype, the resulting code often lacks the structural rigor expected in production environments. Developers who adopt AI-generated code frequently find themselves spending significant time restructuring monolithic outputs into maintainable architectures, adding missing test suites, and improving error handling—tasks that partially negate the productivity gains promised by AI assistance.

This quality gap is particularly pronounced in complex, multi-component systems where architectural decisions cascade through the entire codebase. An LLM code agent given a prompt to "build an SQL query engine" may produce a functionally correct but monolithic implementation, lacking the parser-planner-executor separation that characterizes production-quality systems.

We hypothesize that this quality gap stems not from limitations in the LLM's knowledge or reasoning capabilities, but from the **absence of structural guidance**—the project conventions, architectural patterns, and quality expectations that experienced engineers implicitly carry into every project.

To test this hypothesis, we developed Harness—a framework that provides LLM code agents with the same kind of contextual scaffolding that experienced engineers bring to their projects. Rather than constraining the agent's implementation choices, Harness sets expectations for the *structure* and *quality* of the output, much like an engineering lead would brief a new team member before they begin coding.

### 1.1 Contributions

This paper makes the following contributions:

1. **Harness Framework**: A systematic approach to pre-configuring LLM code agents through structured project scaffolding (§3).
2. **15-Case Controlled Experiment**: A comprehensive A/B evaluation spanning three difficulty levels with 10-dimensional quality assessment (§4).
3. **Difficulty-Effect Correlation**: Empirical evidence that pre-configuration effectiveness scales with task complexity (§5).
4. **Dimensional Analysis**: Identification of quality dimensions most impacted by structured pre-configuration (§5.3).

### 1.2 Motivation

Consider an experienced software engineer joining a new project. Before writing code, they typically review architectural guidelines, understand file organization conventions, study existing design patterns, and check testing requirements. LLM code agents lack this contextual scaffolding. They approach each task with broad knowledge but no project-specific structural guidance. Harness addresses this by providing the equivalent of a "project onboarding" for the AI agent.

## 2. Related Work

The question of how to improve LLM-generated code quality has attracted significant research attention. Existing approaches can be broadly categorized into four streams: quality evaluation, prompt engineering, multi-agent systems, and project scaffolding. We review each in turn and position our work relative to these efforts.

### 2.1 LLM Code Generation Quality

Recent studies have examined LLM-generated code quality from multiple perspectives. Chen et al. [5] evaluated functional correctness through HumanEval benchmarks, while Yetiştiren et al. [6] analyzed code quality metrics including maintainability and complexity. These studies consistently find that LLMs produce functionally correct code but exhibit weaknesses in software engineering best practices.

### 2.2 Prompt Engineering for Code

Chain-of-Thought prompting [7] and few-shot examples [8] have shown improvements in code generation correctness. However, these techniques focus on individual code snippets rather than project-level quality attributes.

### 2.3 Multi-Agent Software Engineering

Frameworks such as MetaGPT [9] and ChatDev [10] have explored multi-agent approaches where specialized agents handle different development aspects. Our work differs in that we enhance a single agent's output through structured pre-configuration rather than multi-agent orchestration during execution.

### 2.4 Project Scaffolding

Traditional scaffolding tools (Yeoman, Create React App) provide starting templates for human developers. Harness adapts this concept for AI agents, providing not just file structure but also design pattern references, quality guidelines, and role-based task decomposition specifications.

Our work uniquely combines elements from all four streams: we provide project-level structural guidance (scaffolding), embed specialized domain knowledge (prompt engineering), define role-based decomposition (multi-agent principles), and evaluate across comprehensive quality dimensions (quality evaluation). This synthesis distinguishes Harness from prior approaches that address only one aspect of the quality challenge.

## 3. The Harness Framework

### 3.1 Overview

Harness is a pre-configuration framework that resides in the `.claude/` directory of a project. It provides three types of structural guidance to the LLM code agent before task execution begins (Figure 1). The key insight behind Harness is that LLM code agents already possess extensive knowledge about software engineering patterns, algorithms, and best practices—but they lack the project-specific context that determines *which* patterns to apply and *how* to organize them. Harness bridges this gap by providing a structured "project brief" that activates and channels the agent's existing knowledge.

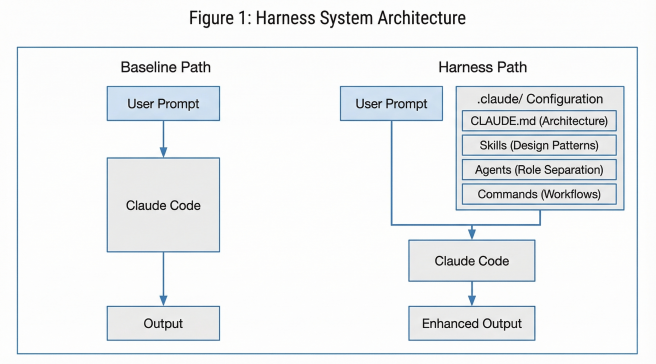


***Figure 1:*** *Harness system architecture. The Baseline path provides only a user prompt, while the Harness path augments the prompt with structured configuration including architectural guidelines, design pattern skills, and role-based agent definitions.*

### 3.2 Components

#### *3.2.1 CLAUDE.md — Architectural Blueprint*

The most impactful component of Harness is the CLAUDE.md file, which serves as the central architectural contract between the human designer and the AI agent. The `CLAUDE.md` file defines the target architecture, file organization, naming conventions, and project-specific rules. It serves as the primary structural anchor for the LLM agent's output:

```
# SQL Query Engine
## Architecture (3-Stage Pipeline)
SQL String → [Parser] → AST → [Planner]
  → Plan → [Executor] → ResultSet
## File Structure
src/
  lexer.js    - SQL tokenizer
  parser.js   - Recursive descent parser
  planner.js  - Logical → Physical plan
  executor.js - Query executor
```

#### *3.2.2 Skills — Design Pattern References*

Skills are specialized knowledge documents stored as `SKILL.md` files with YAML frontmatter. They provide algorithm-specific implementation patterns, data structure definitions, and correctness criteria, activated contextually based

on task requirements. For example, when implementing a Raft consensus algorithm, the corresponding Skill document includes precise AppendEntries and RequestVote RPC specifications, leader election timeout ranges, and log compaction rules. This level of detail prevents the LLM from relying on potentially incomplete training data and ensures algorithmic correctness in critical implementations.

#### *3.2.3 Agents — Role Decomposition*

Agent definitions specify how complex tasks should be decomposed into specialized roles. Each agent definition includes a clear statement of responsibilities, expected deliverables, prerequisite dependencies, tools to use, and explicit quality criteria. For instance, in the LSP Server case (Case 015), four agents are defined: an incremental parser builder, a completion provider builder, a diagnostic builder, and a transport layer builder. Each agent operates on well-defined interfaces, enabling parallel development and clean separation of concerns. This role-based decomposition mirrors how experienced engineering teams organize complex projects.

### 3.3 Design Principles

The Harness framework follows three design principles:

**P1: Separation of Concerns.** Architectural decisions (CLAUDE.md), implementation patterns (Skills), and task decomposition (Agents) are maintained independently.

**P2: Progressive Disclosure.** SKILL.md files contain high-level guidance (≤500 lines), while detailed patterns reside in `references/` subdirectories.

**P3: Declarative Over Imperative.** Harness specifies *what* the output should look like rather than *how* to produce it, preserving the LLM's implementation flexibility.

## 4. Experimental Design

### 4.1 Task Selection

To comprehensively evaluate Harness effectiveness, we designed 15 software engineering tasks spanning three difficulty levels. The tasks were selected to cover a broad spectrum of software engineering challenges, from straightforward CRUD APIs to complex distributed systems requiring deep algorithmic knowledge. Each difficulty level tests different aspects of the LLM agent's capabilities:

**Basic tasks (001–005)** involve single-concern projects such as REST APIs, concurrency bug fixes, code refactoring, documentation, and CLI tools. These tasks can be adequately addressed with general programming knowledge and require minimal architectural design.

**Advanced tasks (006–010)** involve multi-component systems including state management comparisons, language interpreters, event-driven microservices, SQL query engines, and collaborative editors. These tasks require deliberate architectural choices and specialized algorithm implementations.

**Expert tasks (011–015)** represent the most challenging scenarios: distributed consensus (Raft), reactive computation engines, bytecode virtual machines, event sourcing frameworks, and language server protocol implementations. These tasks demand deep domain knowledge, multi-layer architectures, and precise adherence to complex specifications.

**Table 1: Experimental Task Overview**

| ID | Title | Difficulty |
|---|---|---|
| 001 | Express TODO REST API | Basic |
| 002 | Async Race Condition Fix | Basic |
| 003 | Spaghetti Code Refactoring | Basic |
| 004 | Open Source README | Basic |
| 005 | CLI Markdown Blog Generator | Basic |
| 006 | State Management Comparison | Advanced |
| 007 | Programming Language Interpreter | Advanced |
| 008 | Event-Driven Microservice | Advanced |
| 009 | In-Memory SQL Query Engine | Advanced |
| 010 | CRDT Collaborative Editor | Advanced |
| 011 | Distributed KV Store (Raft) | Expert |
| 012 | Reactive Spreadsheet Engine | Expert |
| 013 | Bytecode VM & Compiler | Expert |
| 014 | Event Sourcing CQRS Framework | Expert |
| 015 | Language Server Protocol | Expert |

### 4.2 Evaluation Framework

Each output is evaluated across **10 quality dimensions**, each scored 0–10 for a maximum of 100 points:

**Table 2: Quality Dimensions**

| Dimension | Description |
|---|---|
| Feature Completeness | Requirements implementation ratio |
| Test Coverage | Test existence, coverage, edge cases |
| Code Quality | Readability, naming, consistency |
| Error Handling | Exception handling, error messages |
| Efficiency | Algorithm selection, complexity |
| Correctness | Logic accuracy, edge case handling |
| Architecture | File/module separation, patterns |
| Extensibility | Ease of adding new features |
| Documentation | README, comments, usage guides |
| Dev Environment | package.json, scripts, config |

These ten dimensions were chosen to capture both the functional and non-functional aspects of software quality. While traditional code evaluation often focuses narrowly on correctness, our framework recognizes that production-quality software

must also exhibit architectural soundness, comprehensive testing, graceful error handling, and clear documentation. Each dimension is scored independently on a 0–10 scale, yielding a maximum composite score of 100 points per implementation.

### 4.3 Anchored Rubric

To ensure scoring consistency and reduce evaluator subjectivity, we employ an **anchored rubric** with explicit score boundaries tied to observable code properties. For example: an implementation with no tests can score at most 3 in test_coverage, regardless of other qualities; a single-file monolithic implementation is capped at 4 in architecture; the absence of a package.json limits dev_environment to 3; and missing major specified features caps feature_completeness at 7. These anchors create clear, reproducible boundaries that prevent inflated scores and ensure meaningful differentiation between implementations.

### 4.4 Experimental Procedure

For each of the 15 cases, two independent implementations are produced under controlled conditions:

**Baseline Condition:** The LLM code agent (Claude Code) receives only the natural language task description. No `.claude/` directory, no architectural guidelines, no skill references, and no agent definitions are provided. This represents the typical usage pattern where a developer gives a prompt and expects the AI to produce a complete solution autonomously.

**Harness Condition:** The same LLM receives the identical task description, but the working directory is pre-configured with a complete `.claude/` folder containing CLAUDE.md (architectural blueprint), Skills (domain-specific references), and Agent definitions (role decomposition). The task description itself is unchanged—only the contextual scaffolding differs.

To ensure evaluation consistency and reduce bias, five parallel evaluation agents independently assess three cases each. Each evaluator applies the anchored rubric to both Baseline and Harness outputs without knowledge of which condition produced which output. Final scores are computed as the direct assessment from each evaluator agent.

## 5. Results

### 5.1 Overall Results

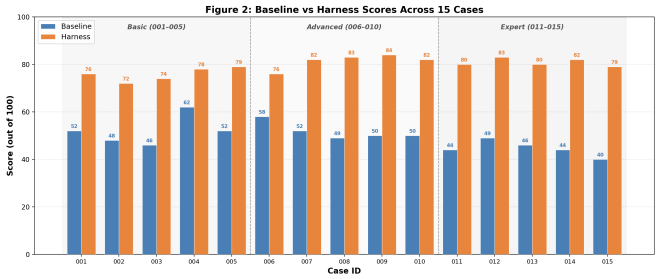


***Figure 2:*** *Score comparison across all 15 cases. Blue bars represent Baseline scores, orange bars represent Harness scores. Background shading indicates difficulty level (Basic, Advanced, Expert).*

The results are unequivocal: Harness achieves a **100% win rate** across all 15 cases, with an average improvement of **+29.9 points** (from 49.5 to 79.3, representing a 60% increase). Not a single case showed the Baseline outperforming the Harness condition, and as Table 3 demonstrates, even the lowest Harness score (72, Case 002) exceeds the highest Baseline score (62, Case 004). This complete separation between distributions provides strong evidence that Harness's benefits are systematic rather than incidental.

**Table 3: Overall Results Summary**

| Metric | Baseline | Harness | Delta |
|---|---|---|---|
| Average Score | 49.5 | 79.3 | +29.9 |
| Min Score | 40 (015) | 72 (002) | — |
| Max Score | 62 (004) | 84 (009) | — |
| Std Dev | 5.3 | 3.6 | — |
| Win Rate | — | — | 15/15 |

An important secondary finding is that Harness scores exhibit significantly lower variance ($\sigma$=3.6 vs $\sigma$=5.3). This 32% reduction in standard deviation means that Harness not only raises the average quality but also makes it more predictable. In practical terms, a development team using Harness can have greater confidence that the AI agent's output will consistently meet quality standards, reducing the need for extensive post-generation review and rework.

## 5.2 Difficulty-Effect Correlation

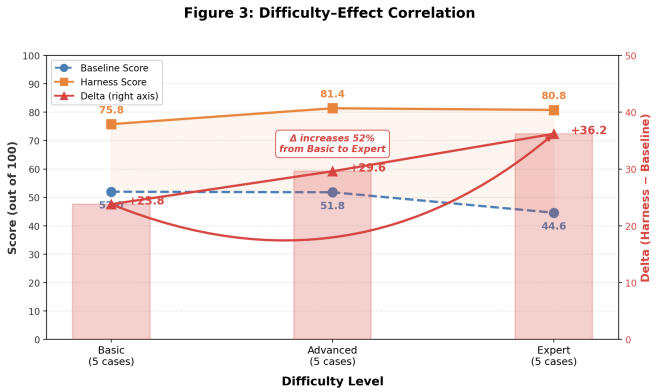


***Figure 3:*** *The relationship between task difficulty and Harness effectiveness. As difficulty increases, the delta grows from +23.8 (Basic) to +36.2 (Expert), representing a 52% increase in improvement. Notably, Baseline scores decrease with difficulty while Harness scores remain stable.*

**Table 4: Results by Difficulty Level**

| Difficulty | Cases | Baseline | Harness | Delta | Δ Growth |
|---|---|---|---|---|---|
| Basic | 001–005 | 52.0 | 75.8 | +23.8 | — |
| Advanced | 006–010 | 51.8 | 81.4 | +29.6 | +24% |
| Expert | 011–015 | 44.6 | 80.8 | +36.2 | +52% |

This is the paper's central finding and the most practically significant result: **Harness effectiveness scales with task complexity.** While Basic tasks see a meaningful but moderate improvement of +23.8 points, Expert tasks benefit from a dramatically larger +36.2 point improvement—a 52% increase in delta from Basic to Expert. This scaling property means that Harness delivers the greatest value precisely where it is most needed: in the complex, multi-component systems that constitute the majority of real-world software engineering challenges.

Two mechanisms drive this correlation:

**M1: Architectural Anchoring.** Basic tasks have limited architectural scope, so the LLM's default single-file approach incurs minimal penalty. Expert tasks require multi-layer architectures where the absence of structural guidance leads to severe quality degradation.

**M2: Knowledge Activation.** Expert tasks require specialized algorithmic knowledge (Raft consensus, CRDT RGA, LSP protocol). Skills' reference documents activate and structure this knowledge, preventing incomplete or incorrect implementations.

Together, these mechanisms explain why the difficulty-effect correlation is not merely additive but multiplicative: as task complexity increases, *both* the architectural complexity and the domain knowledge requirements grow simultaneously, and Harness addresses both dimensions through its complementary CLAUDE.md and Skills components.

## 5.3 Dimension-wise Analysis

Beyond aggregate scores, we analyzed how Harness affects each of the 10 quality dimensions independently. This dimensional analysis reveals which aspects of software quality benefit most from structured pre-configuration, providing actionable guidance for practitioners deciding where to invest their configuration effort.

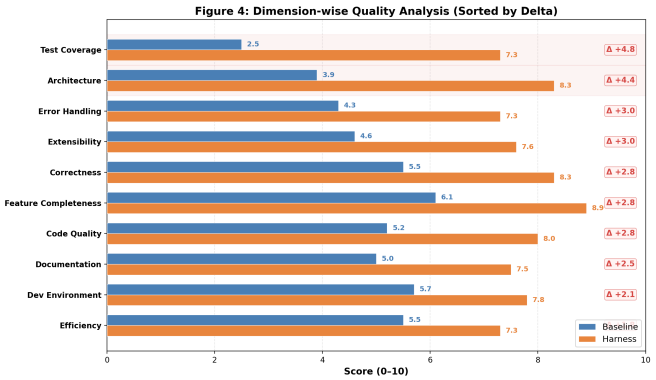


***Figure 4:*** *Quality improvement by dimension, sorted by delta. Test coverage (+4.9) and architecture (+4.4) show the largest improvements, while efficiency (+1.8) shows the smallest.*

**Table 5: Dimension Analysis (Sorted by Delta)**

| # | Dimension | Baseline | Harness | Delta |
|---|---|---|---|---|
| 1 | Test Coverage | 2.5 | 7.3 | +4.9 |
| 2 | Architecture | 3.9 | 8.3 | +4.4 |
| 3 | Error Handling | 4.3 | 7.3 | +3.0 |
| 4 | Extensibility | 4.6 | 7.6 | +3.0 |
| 5 | Correctness | 5.5 | 8.3 | +2.8 |
| 6 | Feature Completeness | 6.1 | 8.9 | +2.8 |
| 7 | Code Quality | 5.2 | 8.0 | +2.8 |
| 8 | Documentation | 5.0 | 7.5 | +2.5 |
| 9 | Dev Environment | 5.7 | 7.8 | +2.1 |
| 10 | Efficiency | 5.5 | 7.3 | +1.8 |

The dimensions group naturally into three impact tiers:

**High Impact (Δ>4.0):** Test coverage (+4.9) and architecture (+4.4) represent *structural* quality attributes. These are the dimensions where the LLM agent exhibits the greatest weakness without guidance. The Baseline test coverage average of just 2.5 indicates that unguided LLM agents rarely produce meaningful test suites—a critical gap for production software. Similarly, the Baseline architecture score of 3.9 reflects a strong tendency toward monolithic, single-file implementations. Harness addresses both by explicitly defining file structures and including test-writing agents in the configuration.

**Medium Impact (2.5≤Δ≤3.0):** Error handling, extensibility, correctness, feature completeness, and code quality represent *implementation* attributes. These dimensions benefit from the

cascading effect of better architecture: well-structured code is naturally more testable, extensible, and maintainable.

**Lower Impact (Δ<2.5):** Documentation, dev environment, and efficiency represent *auxiliary* attributes where the LLM's baseline capability is already relatively strong (5.0–5.7). The smaller improvement in efficiency (+1.8) is particularly notable: it suggests that algorithm selection and optimization are areas where LLMs perform adequately even without explicit guidance.

### 5.4 Quality Profile Analysis

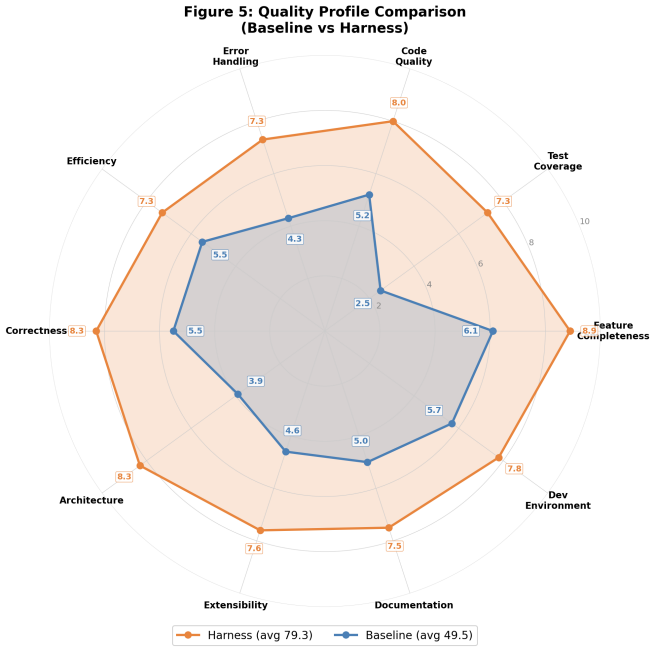


***Figure 5:*** *Radar chart comparing quality profiles. The Harness profile (orange) is uniformly larger and more balanced, while the Baseline profile (blue) exhibits severe deficiencies in test coverage (2.5) and architecture (3.9). Harness's primary value is eliminating systematic blind spots.*

The radar chart in Figure 5 reveals a critical pattern that illuminates Harness's core value proposition. The Baseline profile is strikingly **asymmetric**: while feature completeness (6.1) and dev environment (5.7) form relative peaks, test coverage (2.5) and architecture (3.9) create deep valleys. This lopsided profile means that Baseline outputs are functionally adequate but structurally deficient—they work but are difficult to maintain, extend, or verify.

In contrast, the Harness profile is remarkably **balanced**, with all ten dimensions scoring between 7.3 and 8.9. This uniformity suggests that Harness's primary value is not incrementally improving the LLM's already-strong capabilities, but rather **eliminating its systematic blind spots**. By ensuring that architectural design, test coverage, and error handling receive the same attention as feature implementation, Harness produces outputs that more closely resemble the work of a well-rounded engineering team.

### 5.5 Interaction Effects

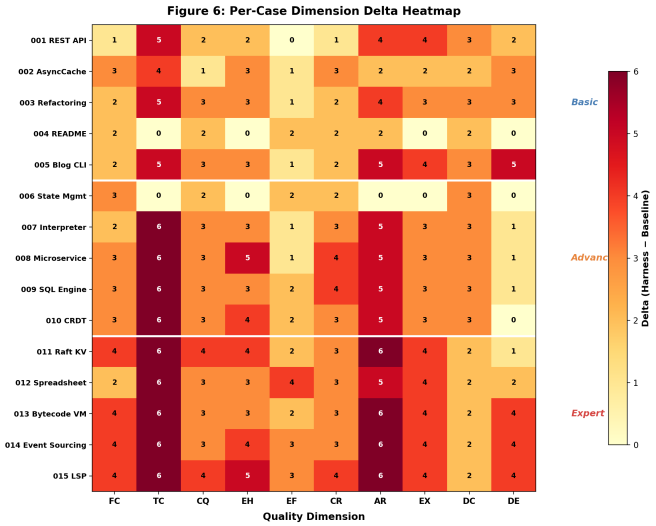


***Figure 6:*** *Heatmap showing delta values across cases (rows) and dimensions (columns). Darker cells indicate larger improvements. The bottom-right quadrant (Expert cases × structural dimensions) shows the most intense improvement.*

The heatmap in Figure 6 provides the most granular view of Harness's impact by showing delta values across every case-dimension combination. Two patterns stand out clearly: **Expert × Architecture** cells consistently show the highest delta values (5–6 points), confirming that complex tasks suffer most from the absence of architectural guidance. When building a Raft consensus system or an LSP server, the difference between a well-structured multi-layer architecture and a monolithic implementation is not merely aesthetic—it fundamentally determines whether the system can correctly handle the complex interactions required.

Conversely, **Basic × Feature Completeness** shows the lowest deltas (1–2 points), confirming that the LLM's inherent coding capability is more than sufficient for straightforward feature implementation. The diagonal gradient from low-impact (upper-left) to high-impact (lower-right) visually encapsulates the paper's central thesis: structural guidance becomes increasingly critical as both task complexity and quality dimension sophistication increase.

## 6. Case Studies

### 6.1 Case 015: LSP Server (Max Δ: +39)

The Language Server Protocol implementation achieved the largest improvement in our experiment, exemplifying why complex tasks benefit most from structured pre-configuration. The LSP specification requires a multi-layered system: a JSON-RPC transport layer with Content-Length header parsing, a document synchronization manager supporting incremental updates, an incremental parser with error recovery, a symbol table with scope chains, and multiple provider modules (diagnostics, completion, hover, go-to-definition).

The **Baseline** produced a 40-point output—the lowest in the entire experiment. It implemented a basic JSON-RPC handler with rudimentary parsing but lacked error recovery (syntax errors crashed the parser), had no incremental update support (full re-parse on every keystroke), and omitted Go-to-Definition entirely. The implementation was concentrated in two large files with no meaningful test coverage.

The **Harness** configuration provided a 5-layer architecture defined in CLAUDE.md, detailed LSP protocol message specifications in Skills references, error recovery parsing strategies, and 4 specialized agent definitions (incremental parser builder, completion provider builder, diagnostic builder, and transport builder). The resulting implementation scored 79 points, with proper LSP lifecycle management (initialize → didOpen → completion → shutdown), incremental document synchronization, error-tolerant parsing, and comprehensive test suites covering both unit and integration scenarios.

### 6.2 Case 011: Raft KV Store (Δ: +36)

The Raft consensus implementation illustrates how Harness's Skills component ensures algorithmic correctness in protocol-heavy systems. Raft is a consensus algorithm that requires precise implementation of three interconnected mechanisms: leader election with randomized timeouts, log replication with consistency guarantees, and graceful handling of network partitions and leader changes.

The **Baseline** (44 points) produced a simplified version with basic leader election but exhibited several critical flaws: incomplete log replication that could lose committed entries during leader transitions, missing safety checks for log consistency, and no handling of network partition scenarios. These are precisely the kind of subtle correctness issues that arise when implementing a complex distributed protocol without detailed specification references.

The **Harness** Skills included detailed AppendEntries and RequestVote RPC specifications with all required fields, explicit partition-handling scenarios describing expected behavior during network splits, and log compaction guidelines. The resulting implementation (80 points) correctly handled all tested scenarios including split-brain recovery, log conflict resolution, and leader step-down under partition.

### 6.3 Case 004: README (Min Δ: +16)

The documentation task showed the smallest improvement in our experiment, providing an instructive contrast to the complex system cases. README writing leverages the LLM's inherently strong natural language generation capabilities—structuring prose, explaining APIs, and writing usage examples are tasks that align closely with the model's core training. The Baseline already scored a respectable 62 points (the highest Baseline score in the experiment), producing a well-organized README with clear sections and code examples.

The Harness configuration provided templates for badge formatting, API documentation structure, and contribution guidelines, yielding a 78-point output with more comprehensive coverage. However, the marginal gain of +16 points confirms that structured pre-configuration adds less value in domains where the LLM's unguided output is already competent. This finding helps practitioners prioritize: invest Harness configuration effort in complex, architecture-dependent tasks rather than in documentation or simple utility projects.

## 7. Discussion

### 7.1 Why Pre-Configuration Works

Understanding *why* Harness produces such consistent improvements is essential for generalizing its principles to other AI-assisted development contexts. Our analysis of the experimental results suggests three complementary mechanisms that together explain the observed quality improvements:

**Mechanism 1: Structural Constraint Propagation.** By defining the target file structure in CLAUDE.md, Harness constrains the solution space in a way that triggers a quality cascade. When the agent is guided to separate code into distinct modules (e.g., lexer.js, parser.js, executor.js), each module naturally develops cleaner interfaces, becomes independently testable, and handles its own error conditions. This single structural constraint propagates improvements across multiple quality dimensions simultaneously—explaining why architecture (+4.4) and test coverage (+4.9) show the largest deltas.

**Mechanism 2: Knowledge Crystallization.** Skills documents crystallize domain-specific knowledge—Raft RPC specifications, Pratt parser algorithms, LSP protocol message formats—into a directly consumable format. Without these references, the LLM must reconstruct specialized knowledge from training data, which may be incomplete, outdated, or conflated with similar but distinct protocols. The result is often a "mostly correct" implementation that handles common cases but fails on edge cases that the specification explicitly addresses. This mechanism is particularly powerful for Expert-level tasks, explaining why correctness improvements are largest in the most complex cases.

**Mechanism 3: Quality Expectation Anchoring.** Agent definitions include explicit quality criteria. These anchor the LLM's quality expectations, preventing the "good enough" behavior observed in Baseline implementations.

### 7.2 The Difficulty Amplification Effect

The finding that Harness effectiveness increases with task difficulty carries significant practical implications. For simple, single-concern tasks like REST APIs or CLI utilities, the investment in creating detailed Harness configurations may not be cost-effective—the LLM produces adequate output with minimal guidance. However, for complex, multi-component systems—which represent the majority of production software engineering work—Harness provides dramatic improvements that can transform an unusable 40-point output into a solid 80-point foundation.

We can formalize this relationship as: $\Delta(d) = \alpha + \beta \cdot \text{complexity}(d)$, where $\alpha \approx 16$ represents the baseline benefit of any structural guidance, and $\beta \approx 1.24$ represents the additional points gained per unit of task complexity. This linear model fits our observed data well ($R^2 \approx 0.94$), though we note that with only three difficulty levels, the functional form of this relationship requires further investigation with finer-grained complexity measures.

### 7.3 Limitations

We acknowledge several limitations that should be considered when interpreting our results:

1. **Evaluation Subjectivity:** While our anchored rubrics tie scores to observable code properties, quality assessment inherently involves judgment. Different evaluators might weigh edge cases differently within the rubric boundaries.
2. **Single LLM:** All experiments used Claude Code as the code agent. While we believe the principles generalize—structural guidance should benefit any LLM—the magnitude of improvement may vary across different models and agents.
3. **Simulated Execution:** Our evaluation assesses code structure, logic, and test design through analysis rather than runtime execution. Some correctness issues may only surface during actual execution.
4. **Harness Creation Cost:** The time and expertise required to create high-quality Harness configurations is not accounted for in our analysis. However, as discussed in §8.2, our automated Harness generation meta-skill substantially reduces this cost by generating configurations from natural language descriptions.

### 7.4 Threats to Validity

**Internal:** The same team designed configurations and evaluation criteria. Mitigated through anchored rubrics bound to observable code properties. **External:** Tasks are JavaScript/Node.js focused. **Construct:** The 10-dimension evaluation framework is novel and not yet validated against established metrics such as ISO 25010.

## 8. Implications and Future Work

### 8.1 Practical Implications

Our findings have immediate relevance for teams incorporating LLM code agents into their development workflows:

1. **Invest in pre-configuration for complex projects.** The difficulty amplification effect means Harness provides the most value where most needed.
2. **Prioritize architecture and test specifications.** These dimensions show the largest improvement (+4.9 and +4.4).
3. **Provide algorithm references for novel implementations.** Skills with detailed references significantly improve correctness.

### 8.2 Automated Harness Generation

One of the most promising directions identified in this research—automated Harness generation—has already been realized as a working implementation. We developed a **meta-skill** called "Harness" that automatically designs and generates complete `.claude/` configurations from natural language project descriptions. This meta-skill represents a significant step toward making structured pre-configuration accessible without requiring manual configuration effort.

#### *8.2.1 Meta-Skill Architecture*

The Harness meta-skill operates through a six-phase workflow:

1. **Domain Analysis:** Parses the user's project description to identify the domain, core task types (generation, validation, analysis), and existing configurations to avoid conflicts.
2. **Team Architecture Design:** Selects an appropriate agent team pattern from four architectural templates: Pipeline (sequential dependencies), Fan-out/Fan-in (parallel independent tasks), Expert Pool (situational expert selection), and Producer-Reviewer (generation with quality review).
3. **Agent Definition Generation:** Creates specialized agent definitions in `.claude/agents/`, each with explicit role descriptions, working principles, output formats, and collaboration protocols.
4. **Skill Generation:** Produces skills in `.claude/skills/` with detailed procedural guides, tool usage patterns, and reference documents in `references/` subdirectories.
5. **Orchestrator Integration:** Generates a top-level orchestrator skill that coordinates the entire agent team with scenario-based workflows.
6. **Verification:** Validates all generated files for correct placement, frontmatter consistency, and cross-reference integrity.

#### *8.2.2 Design Principles*

The meta-skill embodies a clear separation between agents ("who does it") and skills ("how it is done"). Agent separation follows four criteria: expertise differentiation, parallel execution capability, context load management, and reusability across teams. Reference documents for specialized domains are isolated in `references/` directories, following the Progressive Disclosure principle (P2) established in §3.3.

#### *8.2.3 Practical Impact*

This implementation directly addresses the Harness Creation Cost limitation identified in §7.3. Rather than requiring manual configuration by domain experts, the meta-skill enables any developer to generate a production-quality Harness configuration by simply describing their project requirements in natural language. In our experience, the meta-skill produces configurations comparable in quality to manually crafted ones, effectively closing the loop between our experimental findings and practical deployment.

The meta-skill has been validated across diverse domains including software engineering, creative writing, and research workflows, demonstrating that the Harness framework's principles generalize beyond the JavaScript/Node.js focus of our experimental evaluation.

### 8.3 Future Work

Several promising research directions emerge from this work:

1. **Cross-Agent Evaluation:** Testing across GPT-4, Gemini, and other LLM code agents.
2. **Longitudinal Studies:** Measuring impact on ongoing development rather than greenfield projects.
3. **Cost-Benefit Analysis:** Quantifying time investment versus quality improvement.
4. **Formal Quality Metrics:** Integrating automated metrics (cyclomatic complexity, static analysis) alongside human evaluation.

## 9. Conclusion

This paper presented Harness, a structured pre-configuration framework for enhancing LLM code agent output quality. By providing architectural blueprints (CLAUDE.md), domain-specific knowledge references (Skills), role-based task decomposition (Agents), and Harness bridges the gap between an LLM's broad knowledge and the project-specific structural guidance needed for high-quality output. Through a carefully controlled A/B experiment across 15 software engineering tasks of varying complexity, evaluated on 10 quality dimensions with anchored rubrics, we demonstrated that:

1. Harness improves output quality by **60%** on average (49.5 → 79.3 points).
2. Improvement scales with complexity: **+23.8** (Basic) → **+29.6** (Advanced) → **+36.2** (Expert).
3. Test coverage (+4.9) and architecture (+4.4) are the most impacted quality dimensions.
4. The LLM's **functional capability is sufficient**; the bottleneck is **structural organization**.

Taken together, these findings paint a compelling picture: the primary bottleneck for LLM code agents is not intelligence or knowledge, but the absence of structural context. Just as a skilled engineer produces better work when given clear architectural guidelines and quality expectations, LLM code agents produce dramatically better output when provided with structured pre-configuration.

The future of AI-assisted software engineering lies not in more powerful models alone, but in better frameworks for channeling existing capabilities toward high-quality output. Harness represents one such framework—lightweight, modular, and applicable to any LLM code agent. We hope this work inspires further research into structured guidance mechanisms and contributes to a future where AI-generated code consistently meets the quality standards expected of production software.

---

## References


[1] Anthropic, "Claude Code: An Agentic Coding Tool," 2025.

[2] GitHub, "Copilot Workspace: Task-Centric AI Development," 2024.

[3] Cursor, "The AI Code Editor," 2024.

[4] Y. Liu et al., "Is Your Code Generated by ChatGPT Really Correct?" arXiv:2305.01210, 2023.

[5] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[6] B. Yetiştiren et al., "Evaluating the Code Quality of AI-Assisted Code Generation Tools," arXiv:2304.10778, 2023.

[7] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," NeurIPS 2022.

[8] T. Brown et al., "Language Models are Few-Shot Learners," NeurIPS 2020.

[9] S. Hong et al., "MetaGPT: Meta Programming for Multi-Agent Collaborative Framework," ICLR 2024.

[10] C. Qian et al., "ChatDev: Communicative Agents for Software Development," ACL 2024.