Kuwait Journal of Advanced Computer Technology
ISSN: 2984-7907
Vol 4 Issue 1 (2026)

Bridging the Prompt-Code Gap: The Evolving Role of Developers in the Age of Generative AI

Maheswara Rao A V
Author

Submitted: 01/12/2025 **Revised**: 18/12/2025 **Published**: 28/12/2025

Abstract

Generative Artificial Intelligence (GenAI) has rapidly transformed modern software engineering by enabling accelerated code generation, automated debugging, multi-language translation, and architectural prototyping. Despite these capabilities, a persistent disconnect remains between developer intent and AI-generated output—a phenomenon defined in this paper as the **Prompt–Code Gap**. This gap becomes increasingly pronounced in complex, stateful, multi-platform environments such as automated trading systems, where correctness, determinism, and system integrity are critical.

This research provides a comprehensive examination of the Prompt–Code Gap by integrating theoretical analysis with two large-scale, real-world case studies involving TradingView Pine Script strategies, PHP-based cloud routers, MQL5 Expert Advisors, and end-to-end TradingView → Cloud → MT5 execution workflows. Findings reveal that GenAI can accelerate development by up to 10×, yet its outputs frequently suffer from hallucinated functions, platform misunderstandings, regression errors, and incomplete logic—especially when handling financial signals, symbol normalization, state machines, or cross-platform trading logic.

Through an in-depth evaluation of each case study, this paper identifies key failure modes, architectural challenges, and human-AI interaction patterns that shape GenAI-assisted development. It also highlights the indispensable role of human developers in refining logic, designing system architecture, enforcing non-repainting constraints, implementing defensive programming, and integrating platform-specific behaviors. The analysis demonstrates that while GenAI produces 50–70% of scaffolding efficiently, the remaining 30–50%—including correctness, reliability, and domain-specific alignment—still requires human expertise.

This study contributes a structured conceptual framework for understanding the Prompt–Code Gap, outlines best practices for GenAI-assisted engineering, and proposes future research directions aimed at building context-aware, architecture-aligned, and platform-specific AI copilots. The paper concludes that GenAI will not replace software engineers; instead, it will amplify the capabilities of those who understand how to supervise, guide, and integrate AI-generated code into robust, real-world systems.

1. Introduction

Generative AI (GenAI) tools such as ChatGPT and Copilot have transformed software engineering by enabling rapid, conversational code generation across languages and platforms. While these capabilities accelerate prototyping and reduce development effort, their limitations become evident in complex, real-world domains such as financial automation and algorithmic trading.

Modern trading systems integrate multi-language components, distributed services, real-time data flows, and strict platform constraints. In such environments, natural-language prompts often fail to convey full architectural requirements, resulting in a misalignment between intended functionality and AI-generated code—a phenomenon termed the **Prompt–Code Gap**. This gap manifests as logical inconsistencies, incorrect assumptions, platform-specific errors, or silently failing logic, especially in systems requiring deterministic execution and precise cross-platform synchronization.

Although GenAI is highly effective for scaffolding, refactoring, and accelerating development cycles, empirical evidence shows that it struggles with deeper contextual reasoning, multi-platform translation, and state-dependent logic. High-stakes domains such as TradingView Pine Script, PHP signal routers, and MT5 Expert Advisors expose these weaknesses, often requiring substantial human intervention to correct or stabilize AI-generated outputs.

This paper centers the Prompt–Code Gap as a fundamental challenge in AI-assisted engineering and examines it through theory and four detailed case studies. The findings emphasize that despite significant productivity gains, human expertise in architecture, debugging, and verification remains critical for building reliable and scalable automated trading systems.

2. Literature Review

Generative AI has rapidly advanced automated code generation, with studies showing strong performance in producing syntactically correct code, boilerplate structures, and debugging assistance (Peng et al., 2023). Research also reports productivity gains of 30–60% in structured development tasks (McKinsey, 2023). Yet, in complex multi-platform systems, GenAI's limitations become more apparent—especially where stateful logic, strict platform rules, and real-time constraints are involved.

Scholars note frequent hallucinations, missing platform-specific details, and partial solutions that require manual correction (Yetiştiren et al., 2023). LLMs also struggle with long-context consistency and cross-file reasoning (Song et al., 2024), issues that are amplified in financial trading environments where small logic errors can cause significant losses (Banh et al., 2025).

Research on multi-platform translation further shows that GenAI often overlooks critical constraints when moving logic between Pine Script, PHP, Python, and MQL5, forcing developers to adjust or rewrite AI-generated outputs (Castor, 2024). This challenge aligns with the emerging concept of the **Prompt–Code Gap**, which describes the disconnect between developer intent and AI-generated implementation. While LLMs may produce 50–70% of a workable solution, the remaining critical portions rely heavily on human expertise.

Overall, existing studies highlight both the benefits and the inherent limitations of GenAI in high-stakes engineering. Despite accelerating development, human architectural reasoning and domain-specific understanding remain essential—particularly in automated trading systems that demand precision, determinism, and cross-platform coherence.

3. Research Background and Context

The growing use of Generative AI in software engineering and automated trading has reshaped how complex systems are built. As development shifts from manual coding to AI-assisted workflows, engineers must balance theory, empirical behavior, and platform constraints. GenAI models rely on statistical correlations rather than deterministic program reasoning, enabling fast code generation but limiting accuracy in tasks requiring strict sequencing, state handling, or multi-system coordination.

Empirical evidence shows that GenAI performs well on isolated tasks but struggles when integrating with databases, APIs, event-driven workflows, or time-sensitive trading logic. This mismatch—known as the **Prompt–Code Gap**—emerges when AI-generated outputs fail to reflect the developer's full intent or the system's operational context.

In multi-platform trading environments, GenAI must work across Pine Script, PHP/Python middleware, and MQL5—each with distinct execution rules. AI often overlooks these differences, leading to incomplete logic or hallucinated functions. As a result, human supervision remains crucial for handling data types, indexing, real-time execution, synchronization, and safety mechanisms such as duplicate suppression or idempotent processing.

While GenAI accelerates prototyping, it may introduce hidden technical debt if outputs are adopted without verification. Engineers increasingly function as reviewers and integrators, ensuring architectural coherence rather than writing all code manually.

In summary, this condensed analysis highlights GenAI's strengths in speed and scaffolding, its limitations in complex multi-platform systems, and the continued necessity of human architectural judgment—especially in high-precision automated trading environments.

4. The Prompt–Code Gap – Theory and Analysis

4. Theoretical and Empirical Foundations of the Prompt–Code Gap in GenAI-Assisted Engineering

Generative AI has introduced a profound shift in how software systems are conceived, authored, and deployed. While traditional programming relied on explicit instruction, deterministic logic, and strict procedural execution, GenAI enables a higher-level specification of intent through natural language. This shift introduces not only new opportunities but also new challenges, particularly when AI-generated code must interact with complex multi-platform systems such as algorithmic trading infrastructures. This section provides deeply expanded academic analysis and is divided into multiple subsections to reflect theoretical foundations, empirical observations, platform-level implications, and architectural reasoning tied to the Prompt–Code Gap.

4.1 Theoretical Foundations of GenAI in Software Engineering

4.1.1 Token-Based Reasoning vs. Program Execution

LLMs operate by predicting the statistically most probable next token, not by understanding program state or execution semantics. This distinction is subtle yet crucial. When generating code, the model does not evaluate the correctness of the logic, test for edge cases, or reason about the runtime behavior of a program. Instead, it constructs outputs that *look* correct based on patterns seen during training. As a result, while GenAI often delivers code that is syntactically plausible, the generated logic may not align with the intended architecture, platform behavior, or operational constraints. This mismatch forms one of the deepest theoretical roots of the Prompt–Code Gap.

4.1.2 Lack of Stateful System Understanding

Software systems evolve through time: they maintain state, accumulate data, interact with external services, and follow deterministic workflows. LLMs have no persistent internal state between prompts; each request is processed independently unless context is manually provided. In system engineering, however, proper functioning depends on sequential actions, message ordering, transaction handling, or cross-module interactions. Without native memory or execution tracing, GenAI cannot fully model or reason about state transitions. Thus, it struggles with orchestrating end-to-end workflows such as trade execution sequences, candle-by-candle indicator evaluation, or multi-step database operations.

5.1.3 Hidden Assumptions and Implicit Architectural Knowledge

Human developers often rely on mental models that capture design principles, system history, domain rules, and business logic. These assumptions are rarely articulated explicitly in prompts. GenAI, lacking access to such tacit knowledge, infers intent based on approximations drawn from training data. This often leads to oversimplification or misinterpretation of architectural needs. For instance, the AI may assume an API exists because similar APIs appear in training examples, even though the specific platform does not support them. Such inferred assumptions widen the Prompt–Code Gap and necessitate human correction.

4.2 Empirical Observations from Real-World Engineering

4.2.1 Partial Correctness and Incomplete Outputs

In real-world implementation environments, AI-generated code frequently achieves partial correctness. For tasks like boilerplate generation, syntax correction, or function stubs, GenAI performs exceptionally well. However, when tasks involve nuanced logic, multi-module integration, or real-time constraints, the outputs begin to degrade. Developers often find that while the model completes 50–70% of the code, the remaining portion—usually critical edge cases or architectural logic—must be manually authored. This pattern is consistent across web development, API design, Pine Script indicators, and MQL5 Expert Advisors.

4.2.2 Iterative Prompt–Repair Cycles in Development

A consistent empirical pattern across all case studies is the iterative nature of AI-assisted coding. Developers begin with an initial prompt, receive a partial solution, refine it, and then prompt for corrections. Instead of converging linearly, AI sometimes introduces regressions—breaking logic that previously worked. This creates a cycle where human developers must continually validate outputs, track inconsistencies, and preserve architectural decisions that the AI forgets over time. These "prompt–repair cycles" illustrate that AI is not a deterministic collaborator but a probabilistic generator requiring ongoing supervision.

4.2.3 Hallucination of Nonexistent or Invalid APIs

One of the most documented empirical phenomena in GenAI models is hallucination—the creation of APIs, functions, or capabilities that do not exist. In the domain of trading systems, hallucination is frequent when the model assumes that Pine Script can read user-drawn trendlines or that MQL5 allows dynamic object introspection. When confronted with underspecified prompts, AI often produces outputs that conform to generalized expectations rather than platform constraints. This behavior increases complexity, as developers must identify and remove hallucinated code before proceeding.

4.2.4 Loss of Long-Range Context and Design Consistency

Long development sessions often reveal a tendency for AI to "forget" earlier decisions. Variable names drift, logic conventions change, and constraints provided in previous prompts are sometimes ignored in later responses. This inconsistency makes it difficult to scale AI-generated code into multi-module architectures or systems requiring strict interface contracts. Developers must therefore preserve and reintroduce context continuously, which introduces cognitive overhead and highlights another dimension of the Prompt–Code Gap.

4.3 Multi-Platform Engineering Implications

4.3.1 Differences in Execution Models Across Platforms

Real-world trading systems consist of multiple platforms—each with a distinct execution model. Pine Script is executed on every bar in a non-looping, immutable environment; cloud routers operate statelessly and rely on REST semantics; and MQL5 executes tick-by-tick with synchronous order handling. These differences are non-trivial and must be respected in system design. GenAI often fails to model or differentiate such execution paradigms, leading to code that appears correct but behaves unexpectedly. For example, logic that works in Pine Script may fail when ported to MQL5 due to differences in indexing and bar formation.

4.3.2 Type Systems, Indexing Models, and Data Constraints

Platforms such as PHP, Python, Pine Script, and MQL5 each enforce their own internal rules regarding typing, memory, and data referencing. LLMs frequently misjudge these rules when translating code across languages. Off-by-one errors, mismatched array bounds, incorrect struct references, or null-handling discrepancies commonly emerge. These issues are amplified in financial systems, where even minor deviations can alter trade timing or risk exposure.

4.3.3 Practical Constraints That AI Cannot Infer Automatically

Several constraints are invisible to AI unless explicitly stated. For instance:

- Pine Script **cannot** access user-drawn drawings.
- MQL5 **requires** pre-defined buffers for indicators.
- MT5 **rejects** orders during invalid symbol sessions.
- PHP routers **must** maintain idempotency to prevent double execution.

GenAI's inability to infer these constraints autonomously underscores the need for human architectural oversight.

4.4 Technical Challenges in GenAI-Assisted Trading System Development

4.4.1 Real-Time Constraints and Latency Sensitivity

Automated trading systems operate under stringent real-time requirements. Trade execution timing is influenced by market volatility, liquidity fluctuations, and millisecond-level latency variations between TradingView, cloud servers, and MT5 terminals. GenAI-generated code frequently overlooks these time-sensitive constraints because the model is unable to simulate network delays, order routing times, or execution queues. As a result, AI may generate logic that is theoretically correct but practically misaligned with the temporal behavior of financial markets, such as assuming immediate order confirmations or failing to handle delayed webhook deliveries. Developers must manually implement rate limiting, timeout handling, and retry logic to ensure resilient execution across distributed components.

4.4.2 Determinism vs. Probabilistic Output

AI-generated code is inherently nondeterministic — different prompts, or even identical prompts, can yield different outputs. This lack of determinism conflicts with the requirements of trading system engineering, where reproducibility, predictability, and deterministic behavior are essential. A trading strategy must execute identically across sessions and platforms; even slight deviations in logic can produce different trade entries or exits. The introduction of probabilistic behavior through AI-generated variability requires developers to enforce strict versioning, audit trails, and code freezes to maintain system integrity. Without these human-led safeguards, AI-enabled pipelines risk producing inconsistent behavior in production environments.

4.4.3 Error Propagation and Silent Failures

One of the most dangerous aspects in the context of automated trading systems is silent failure — a scenario in which the system appears to function normally but does not execute intended trades. AI-generated code is susceptible to subtle logical errors such as incorrect symbol mappings, faulty variable scoping, or improper state resets, which can propagate through multiple layers of the system. These issues often remain undetected until a missed trade or unintended execution occurs in live markets. Human developers must implement multiple layers of logging, telemetry, and validation to detect such conditions proactively. GenAI cannot autonomously design such fail-safe mechanisms because it lacks contextual awareness of financial risk and operational expectations.

4.4.4 Data Integrity and Synchronization Challenges

Trading systems depend heavily on data consistency across various components — price feeds, webhook signals, database tables, pending trade queues, and MT5 order histories. AI-generated code occasionally mishandles data synchronization, such as inserting duplicate records, failing to delete stale entries, or misaligning database schema relationships. These shortcomings are particularly evident in complex CRUD operations involving trades, where race conditions or incomplete updates can compromise system reliability. Developers must therefore architect robust synchronization logic, transactional safeguards, and atomic update mechanisms to guarantee consistency — tasks that remain beyond the autonomous reasoning capability of current GenAI models.

4.5 Architecture-Level Reasoning and System Design Limitations

4.5.1 Distributed System Complexity Exceeds AI Inference Capabilities

Even advanced LLMs struggle to model the implicit architecture of a distributed trading system. A typical pipeline involves request–response cycles across TradingView, cloud PHP routers, MySQL databases, and MT5 Expert Advisors. AI-generated outputs rarely capture the interplay between these components, such as order status transitions, synchronization between pending and executed trades, or management of asynchronous callbacks. Human architects must define these workflows explicitly, as AI tends to produce localized solutions without considering system-wide consequences. This is one of the strongest manifestations of the Prompt–Code Gap: the gulf between local code generation and global architectural alignment.

4.5.2 Human-Only Capabilities in Domain-Specific Reasoning

Certain design decisions require deep domain familiarity that LLMs cannot replicate, such as:

- selecting between bar-close and intrabar evaluation,
- determining optimal buffer indexing for Heiken Ashi calculations,
- designing position-management rules in MT5,
- enforcing idempotency in webhook processing,
- resolving symbol-mismatch issues (e.g., USTEC vs. USTECm),
- tuning performance for Pine Script execution on high-frequency charts.

These choices are not merely technical; they are strategic decisions based on the developer's experience of market behavior, platform limitations, and risk management principles. AI can assist in writing supporting code but cannot autonomously make such decisions due to lack of experiential reasoning.

4.5.3 Lack of Awareness of Non-Functional Requirements (NFRs)

While AI can generate functional code, it does not inherently account for non-functional requirements such as security, scalability, performance, auditability, or maintainability. For instance:

- A trading system must be secure against repeated webhooks, replay attacks, or invalid API keys.
- It must scale to multiple terminals and symbols without degrading performance.
- It must provide traceable logs for compliance and testing.

AI-generated code typically omits these considerations unless explicitly instructed. In contrast, human architects embed these constraints throughout the system design process, demonstrating areas where AI's applicability is limited.

4.5.4 Bridging Logic: The Final 20–40% Only Humans Can Provide

A consistent theme across all real-world case studies is the necessity for human-authored "bridge code." This refers to the custom logic that stitches together different modules, enforces business rules, handles exceptions, and maintains consistent system state. Examples include:

- clearing stale pending trade rows,
- interpreting double signals from TradingView,
- recalculating trailing stop loss based on platform-specific rules,
- adjusting indicator computations to match cross-platform behavior.

This bridge code is rarely generated correctly by AI because it resides at the intersection of domain knowledge, architectural design, and empirical behavior observed during live testing.

4.6 Cross-Platform Translation and Semantic Alignment

4.6.1 Semantic Drift in Code Conversion Across Languages

When AI translates logic across languages — for example, from Pine Script to MQL5 — semantic drift frequently occurs. This refers to subtle changes in meaning or behavior introduced unintentionally through translation. While variable names and structures may appear correct, execution semantics differ significantly due to platform architecture. For example, Pine Script recalculates full history on every bar, while MQL5 handles data incrementally. These differences cause logical drift, resulting in signal mismatches, misaligned calculations, or inconsistent order triggers. Human intervention is required to validate, compare, and calibrate logic so both platforms yield identical outcomes.

4.6.2 Event-Driven vs. Declarative vs. Imperative Models

TradingView Pine Script is declarative; MT5 is event-driven; PHP routers are procedural. GenAI struggles to adapt logic between these paradigms because it lacks an internal model of control flow. Attempts to convert strategies often lead to logical errors such as calculating indicators before buffer initialization or referencing future bars implicitly. Human engineers must therefore redesign the logic flow manually to respect platform constraints, demonstrating that code translation is not purely syntactical but fundamentally architectural.

4.6.3 Handling Platform Constraints and Execution Contexts

Each platform imposes strict constraints:

- Pine Script forbids dynamic loops and memory allocation.
- MQL5 requires preallocated indicator buffers.
- PHP executes statelessly across requests.
- TradingView webhooks cannot be delayed or retried.

AI-generated solutions often violate these rules by suggesting unsupported APIs or invalid operations. This leads to runtime errors or platform rejections. Experienced developers must rewrite or constrain logic to fit within these constraints, further widening the Prompt–Code Gap.

4.6.4 Achieving Cross-Platform Fidelity in Trading Logic

Ensuring that a strategy behaves identically across platforms is one of the most demanding challenges. During the case studies, identical Heiken Ashi formulas, breakout models, and trailing stop logic often produced different results due to differences in timing, tick arrival, bar formation, and execution sequencing. Achieving a 99% cross-platform match required manual inspection of charts, sample-by-sample comparison, and iterative refinement — a task far beyond the autonomous abilities of GenAI.

4.7 Synthesis: Understanding the Prompt–Code Gap as a Central Engineering Limitation

4.7.1 AI Accelerates Development but Cannot Own Architecture

The real-world evidence from all case studies confirms that AI is invaluable in accelerating development, generating boilerplate code, and producing quick drafts. Yet, architecture — the foundation of any large-scale system — cannot be reliably generated by LLMs. Architecture requires holistic awareness of external systems, operational constraints, and domain knowledge, which AI lacks.

4.7.2 Human Expertise as the Governing Layer

In every case, the human developer acted as the supervisor, corrector, designer, and system integrator. AI served as a high-speed collaborator, but human reasoning provided the final direction, corrections, and validations necessary for stable deployment. This symbiosis represents the long-term future of software engineering.

4.7.3 Why the Prompt–Code Gap Will Continue to Exist

The Prompt–Code Gap persists because AI lacks:

- experiential intuition,
- real-time feedback from system execution,
- awareness of risk or financial consequences,
- architectural understanding,

- stateful reasoning capabilities.

Until future models incorporate deeper semantic reasoning and dynamic simulation capabilities, this gap will remain a defining engineering constraint.

4.7.4 Implications for GenAI-Assisted System Design

Developers must treat AI as a cognitive accelerator — not an autonomous engineer. Proper usage involves:

- carefully crafted prompts,
- verification loops,
- modular integration,
- strong validation pipelines,
- human-authored bridge code.

This epistemological shift positions developers not as coders but as "AI supervisors" and "architectural orchestrators."

5. Case Study 1 – Engineering a Fully Automated Trading Pipeline Using TradingView, Cloud Webhooks, and MT5

---

1. Overview, Background, and Problem Context

This case study analyzes the accelerated development of a fully automated execution pipeline connecting TradingView, a cloud-based PHP routing engine, and the MetaTrader 5 (MT5) execution terminal. The entire system—typically requiring several weeks of architecture design, coding, and platform integration—was completed within approximately ten hours using intensive Generative AI assistance. Automated trading environments impose stringent requirements on latency, state synchronization, and platform-aware logic, making them a strong testbed for evaluating the capabilities and limitations of GenAI. While TradingView provides signals, the cloud router performs validation and dispatch, and MT5 executes trades, the system must maintain coherence across asynchronous layers. The project revealed that GenAI could rapidly generate foundational code structures but struggled to maintain consistent logic across languages, schemas, and platform constraints. The primary objective was to create a production-ready system capable of validating and normalizing webhook alerts, enforcing per-symbol idempotency, suppressing duplicate or stale trade instructions, managing a robust pending-trade queue, and ensuring strict synchronization between cloud processing and MT5 execution cycles. The development process highlighted the Prompt–Code Gap, showing that while AI accelerates code generation, human expertise remains essential for ensuring architectural stability and cross-platform consistency.

---

2. System Architecture Overview

The automated trading pipeline consisted of three tightly integrated layers. The TradingView strategy served as the signal-generation component, producing BUY and SELL alerts accompanied by structured JSON payloads. These alerts were delivered to a PHP-based cloud router responsible for validating API credentials, normalizing symbol formats to MT5-compatible naming rules, suppressing duplicate or stale signals, and inserting validated trade instructions into a pending-trades queue backed by MySQL. The router also maintained comprehensive event logs and execution histories. The execution layer resided in MetaTrader 5 through an Expert Advisor that continuously polled the pending queue at high frequency. Upon receiving a new trade instruction, the EA executed the order atomically, applied the required stop-loss and take-profit parameters, and returned a callback to the router to confirm execution. The system further preserved execution integrity by preventing repeated or unintended trades, ensuring that each signal resulted in a single, deterministic execution. Collectively, the layered architecture formed a low-latency, closed-loop trading engine capable of operating continuously around the clock.

3. Pipeline Operations

The operational workflow followed a compact but highly coordinated sequence. TradingView triggered a webhook containing the breakout or directional alert. The cloud router validated the payload, authenticated the request, and applied symbol-normalization rules before purging any stale or duplicate instructions associated with the same symbol. After validation, the router inserted the request into the pending-trades table. The MT5 Expert Advisor polled the queue every 100–300 milliseconds, retrieved the next eligible trade, executed the order, and immediately sent a callback with execution details. The router then updated the trade history and maintained state consistency for subsequent alerts. This streamlined sequence provided deterministic signal flow and sub-second execution latency under normal conditions.

4. AI-Assisted Development Process

The development process relied heavily on Generative AI for generating the structural components of the system. AI produced initial drafts of webhook endpoints, JSON handlers, database schemas, polling loops, order-execution routines, and duplicate-suppression logic. Despite this acceleration, numerous logical gaps surfaced. AI frequently misinterpreted schema relationships, forgot earlier instructions, generated syntactically correct but logically inconsistent code, and introduced regressions during iteration. Human developers performed extensive refinements, including correcting SQL joins, stabilizing symbol-normalization logic, restructuring buffer management within the EA, resolving race conditions, and enforcing idempotency. Through systematic review and correction, human oversight increased the reliability of AI-generated components from roughly 50% to approximately 90%, demonstrating GenAI's value primarily as a scaffolding generator rather than a complete system designer.

5. Technical Challenges

The project encountered several challenges attributable to the limitations of GenAI in multi-layer system development. The AI was unable to maintain clean transitions across pending, executed, and cleared trade states, resulting in inconsistent state synchronization. TradingView occasionally produced duplicate alerts, which AI-generated logic failed to handle robustly, necessitating manual symbol-level cleanup rules. Polling loops created by AI suffered from edge-case issues such as off-by-one errors, stale queue retrieval, and insufficient timeout handling. Symbol normalization proved especially problematic, as the AI repeatedly forgot mapping conventions (e.g., transforming USTEC to USTECm). Database duplication issues also emerged because AI-generated queries preserved outdated records, making manual implementation of strict delete-before-insert rules necessary to ensure idempotent trade processing. These challenges illustrated the gaps between AI-generated scaffolding and the rigor required for a real-time trading environment.

6. Architectural Decisions and Human Intervention

Several human-led architectural decisions were essential to achieving a stable production system. The cloud router was intentionally designed to remain stateless, delegating all persistent state handling to MT5, which ensured deterministic execution. Symbol-level idempotency was enforced to guarantee that only one pending trade existed per instrument at any time. An aggressive purge logic was introduced to eliminate stale or duplicated alerts prior to queue insertion. Poll-based execution was selected because MT5 lacked efficient push-based communication mechanisms. Human engineers also authored crucial bridge code, including schema refinements, comprehensive duplicate-suppression functions, robust state-machine transitions, regression corrections, and custom race-condition protection logic. Approximately 30–40% of the final system consisted of manually implemented logic required to bind AI-generated components into a coherent, reliable production pipeline.

---

7. Outcomes and Performance

The completed system successfully automated the full TradingView-to-MT5 trading workflow, achieving stable sub-second execution latency and eliminating all instances of duplicate trades. The pipeline sustained continuous 24×7 operation with consistent behavior across all layers, supported by comprehensive logging and auditability. The collaboration between AI-generated scaffolding and human architectural oversight resulted in a highly efficient development process, compressing what traditionally requires 15–30 days of engineering effort into roughly ten hours. This outcome demonstrates the effectiveness of hybrid AI-assisted development, where GenAI accelerates implementation while human expertise ensures correctness, resilience, and platform-aligned execution.

6. Case Study 2 – Multi-Platform Parallel-Channel Breakout System Using GenAI, Pine Script, Cloud Routing, and MT5

---

1. Overview, Background, and Problem Context

This case study examines the development of a fully automated parallel-channel breakout trading system that integrates TradingView for strategy logic, a PHP-based cloud router for signal validation, and MetaTrader 5 (MT5) for execution. The project demonstrates how Generative AI can accelerate multi-platform development by generating initial code components, while also revealing the limitations of AI when applied to complex, stateful trading systems requiring high determinism, cross-platform consistency, and strict timing control. Parallel-channel breakout systems depend on stable channel geometry, slope coherence, and multi-factor confirmation involving Renko directional alignment, ATR-based volatility validation, and volume participation thresholds. Automating such a strategy across three heterogeneous platforms required AI to maintain consistent logic across Pine Script, PHP, and MQL5—an area where discrepancies frequently emerged, highlighting the Prompt–Code Gap. The goal of the project was to produce a reliable system capable of detecting breakout events, confirming them with supporting indicators, transmitting deduplicated alerts, validating them on the cloud, and executing trades through an MT5 Expert Advisor while maintaining full state synchronization.

---

2. System Architecture Overview

The automated breakout system was implemented as a coordinated, multi-platform pipeline. On TradingView, a Pine Script strategy calculated channel boundaries, monitored slope stability, detected breakout events using candle-close rules, and applied Renko, ATR, and volume filters before issuing a JSON-formatted webhook. The cloud router received this alert, authenticated it, normalized the symbol to match MT5 naming conventions, revalidated the breakout logic server-side, suppressed duplicate or stale signals, and inserted confirmed breakout events into a pending-trades queue while maintaining a full audit trail. MT5 handled the execution layer through an Expert Advisor that polled the pending queue in short intervals, executed trades atomically, set stop-loss and take-profit parameters, and returned callback confirmations to the cloud. A dedicated state-synchronization layer maintained information on the last breakout direction, channel validity, and recent execution timestamps to prevent unwanted re-entries into the same breakout leg. Together, these layers formed a closed-loop system capable of transforming TradingView breakout detections into consistent, validated, and executable trading actions on MT5.

---

3. Pipeline Operations

The operational flow of the system followed a tightly coordinated sequence. TradingView first generated channel geometry and verified breakout conditions based on directional closes, Renko trend alignment, ATR thresholds, and volume filters. Once a valid breakout was detected, the strategy emitted a webhook containing all relevant parameters. The cloud router parsed the payload, validated its integrity, normalized the symbol, eliminated duplicates, and queued the breakout for execution. The MT5 Expert Advisor then polled the queue at high frequency, retrieved the next eligible trade, executed it, and immediately communicated the execution status back to the router. The router updated the trade history and synchronized the state-tracking variables to ensure the next breakout event was processed correctly. This coordinated pipeline provided a seamless end-to-end flow with sub–200 ms latency in most test scenarios.

4. AI-Assisted Development Process

Generative AI played a central role in rapidly producing initial drafts of the TradingView strategy, PHP middleware, and MQL5 execution logic. It generated Pine Script code for detecting channel geometry, validating breakout conditions, and preparing webhook alerts. For the cloud router, AI provided endpoint templates, JSON parsers, symbol-mapping logic, and MySQL integration code. For MT5, it produced polling loops, order-sending functions, and basic deduplication routines. However, AI-generated code revealed inconsistencies across platforms. Logical discrepancies such as mismatched variable names, missing edge-case conditions, incorrect slope handling, and incomplete confirmation criteria surfaced repeatedly. AI also made incorrect assumptions about Pine Script's drawing model, PHP's array-handling semantics, and MQL5's strict typing. Human intervention was required to realign logic across all platforms, correct execution rules, restore consistency, and ensure compatibility with real-time trading requirements.

5. Technical Challenges

Several technical challenges emerged due to GenAI's inability to maintain logical consistency across heterogeneous environments. Channel geometry detection became unstable as the AI struggled to reproduce identical boundary and slope rules between Pine Script and PHP. Breakout confirmation logic was often incomplete, with AI omitting Renko alignment, ATR thresholds, or volume constraints. Symbol normalization repeatedly broke, as GenAI forgot mapping conventions between TradingView and MT5. Duplicate breakout alerts presented further difficulties because AI-generated logic could not reliably suppress intrabar signals or purge stale queue entries. State synchronization also proved problematic; AI failed to track breakout direction, manage reversal scenarios, or prevent repeated entries during the same trend leg. Human engineering provided the corrective framework, including custom state machines, complete breakout-validation matrices, stable symbol-mapping rules, and robust duplicate-suppression algorithms.

6. Architectural Decisions and Human Intervention

To ensure system stability, several architectural decisions were made manually. Unstable channels with insufficient age or excessive slope deviations were rejected. Breakouts were validated using a five-factor confirmation matrix that assessed boundary interaction, Renko direction, ATR volatility, volume confirmation, and structural alignment with previous swing points. Cross-layer validation was adopted as a mandatory rule, requiring every breakout to pass checks in Pine Script, PHP, and MQL5 before execution. Human engineers rewrote channel geometry functions, corrected drawing logic in Pine Script, rebuilt symbol-mapping rules in PHP, and reconstructed the MQL5 execution engine with atomic execution, retry loops, callback idempotency, and precise order sizing. These manual contributions—representing approximately 30–40% of the final codebase—were essential in converting fragmented AI outputs into a reliable multi-platform system.

7. Outcomes and Performance

The final system achieved real-time, end-to-end breakout detection and execution with sub–200 ms latency. Duplicate signals were reduced by more than 98%, and the Renko confirmation layer significantly improved breakout accuracy. The system maintained stable synchronization across TradingView, PHP middleware, and MT5, ensuring that each breakout event translated into a single, precise execution. Despite the heavy use of Generative AI in the initial development phase, human architectural oversight ultimately ensured system integrity, consistency, and production-grade reliability.

7. Discussion

The findings across all case studies reveal that the integration of Generative AI into large-scale automated trading systems introduces a unique blend of acceleration and complexity. While GenAI significantly reduces development time by generating code templates, structural modules, and strategy prototypes, its limitations become increasingly evident when the system must operate across heterogeneous platforms such as TradingView, PHP middleware, and MT5 execution environments. The core challenge emerges from the **Prompt–Code Gap**, where AI-generated outputs lack precise temporal reasoning, state awareness, and cross-platform contextual understanding. This gap is especially visible in recursive calculations, multi-timeframe synchronization, and logic that depends on bar-based transitions or live feed behavior—areas where GenAI models often make deterministic but incorrect assumptions.

Empirical observations further demonstrate that automated trading workflows require high-fidelity control over candle formation rules, execution timing, latency parameters, and error suppression mechanisms. These are deeply intertwined with platform-specific behaviors, making them difficult for GenAI to infer without explicit architectural guidance. Multi-platform engineering thus becomes a layered problem: signals must remain consistent across TradingView's bar-indexed environment, PHP's stateless HTTP layers, and MT5's tick-driven EA execution cycle. GenAI can assist in drafting components of this pipeline, but the orchestration, state synchronization logic, and error-handling framework still require human-led architectural reasoning.

Additionally, the case studies highlight the importance of **human–AI collaborative workflows**. When human architects validate, refine, or override GenAI-generated components, system reliability increases dramatically. This hybrid development model not only mitigates logic drift but also enhances maintainability by enforcing consistent design rules across layers. The insights indicate that GenAI is most effective as a force multiplier—accelerating code generation and ideation—while human expertise remains essential for correctness, stability, and domain-specific optimization. Overall, the discussion underscores that successful deployment of GenAI-assisted trading systems depends on synergistic collaboration between AI automation and human architectural oversight.

8. Open Research Areas and Future Studies

Several research gaps remain in the use of GenAI for multi-platform automated trading. A key challenge is developing **cross-platform consistency mechanisms** that allow AI models to maintain identical logic across Pine Script, PHP, Python, and MQL5 without drifting or introducing regressions. Future work should also focus on **context-stable AI models** that can retain long-term architectural constraints such as state behavior, timing rules, and execution dependencies.

Another important direction is the creation of **AI-driven validation and simulation frameworks** capable of detecting inconsistencies, race conditions, or duplicate-signal risks before deployment. Integrating GenAI with **reinforcement learning, agentic workflows, and market digital twins** presents further opportunities for improving strategy robustness.

Overall, future studies should explore **hybrid autonomous architectures** in which GenAI handles rapid code generation while human-designed oversight layers enforce determinism, safety, and regulatory compliance. This approach offers the most realistic path toward reliable GenAI-assisted trading systems.

9. Conclusion and Future outlook

This research shows that Generative AI is a powerful accelerator for building multi-layer automated trading systems, but it remains limited in handling the architectural depth, platform constraints, and deterministic logic required for real-world financial execution. While GenAI efficiently generates scaffolding, templates, and initial strategy logic, it struggles with cross-platform consistency, timing precision, and state management—revealing a persistent Prompt–Code Gap between AI-generated code and production-grade requirements.

Human expertise is still essential for designing execution pipelines, aligning TradingView → Cloud → MT5 logic, managing bar transitions, enforcing idempotency, and mitigating race conditions. GenAI can speed up development, but it cannot independently ensure correctness, reliability, or risk-aware behavior.

Looking forward, future GenAI systems may offer deeper architectural memory, cross-language consistency checks, and platform-aware reasoning to reduce manual intervention. However, the most effective path remains a hybrid model where AI accelerates creation and humans ensure structural integrity. In this collaborative framework, GenAI functions as a force multiplier—enabling faster iteration, improved system quality, and more scalable trading automation when guided by experienced developers.

References

Peng et al., 2023 — https://arxiv.org/abs/2302.06590

Yetiştiren et al., 2023 — https://arxiv.org/abs/2304.10778

Pandey et al., 2024 — https://arxiv.org/abs/2406.17910

Banh et al., 2025 — https://www.sciencedirect.com/science/article/pii/S0950584925000904

Song et al., 2024 — https://arxiv.org/abs/2410.02091

McKinsey, 2023 — https://mckinsey.com

Codacy, 2024 — https://blog.codacy.com