International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Generative AI for Data Engineering: A Seven-Stage Orchestration Framework for LLM-Powered Code Generation

Mosaic Basha Syed

Abstract: Data engineering organizations have encountered difficulties with productivity, with platform complexity and the requirement to use multiple technologies, programming languages, and frameworks increasing the effort required to develop data pipelines. Maintaining existing pipelines is challenging and costly with changing requirements, modernization, and poor documentation relative to implementation, complicating the transfer of knowledge and debugging. We propose a seven-stage orchestration architecture to apply LLMs in enterprise data engineering workflows to close the divide between LLMs' theoretical code generation capabilities and practical deployments of such systems in strictly regulated environments. The architecture implements a process that leverages specification ingestion, retrieval augmented generation (RAG), multi-stage code generation with semantic validation, auto documentation writing, multi-layer security scanning, confidence-gated human-in-the-loop review, CI/CD deployment, and reinforcement feedback-based continuous learning to govern the LLMs. We adopt enterprise guardrails like data classifications, metadata-only retrieval, generation scope limits, and immutable audit trails to ensure security, regulatory compliance, and motivated assurance. Recent article are in code generation literature show that multi-turn synthesis, bidirectional context modeling, and human feedback can substantially improve generation effectiveness, which informs our design choices. We propose a thorough architecture for responsibly deploying LLMs for enterprise data engineering and plan to validate this approach with production deployment of LLMs in banking data platforms.

Keywords: *Generative AI, Large Language Models, Code Generation, Data Engineering, Orchestration Framework, Enterprise Architecture*

1. Introduction

Enterprise data platforms almost always contain thousands of ETL pipelines, batch processes, streaming applications, and analytics workflows built on top of various technologies such as Ab Initio, Informatica, Apache Spark, Airflow, AWS Glue, Snowflake, and others. The system engineers responsible for building and maintaining these pipelines would need deep knowledge of data modeling, SQL, Python, Java, Scala, distributed systems, cloud providers and domain specific tools. Engineers spend a lot of time repeating boilerplate, duplicating documentation, testing, debugging, and adding extensions for each platform. Knowledge silos emerge, with only a single engineer having expertise in a system, and getting new team members up to speed takes time. This limits how quickly organizations can deliver data capabilities [1].

Recently, large language models have achieved strong results for code generation. For example, CodeGen models achieve a pass@100 score of 75.00 on the HumanEval benchmark, meaning that the model is expected to solve the task correctly at least once in 100 attempts [1]. Multi-turn program synthesis achieves a pass@100 score of 47.34, compared to 38.74 for single-turn synthesis, by breaking task specifications into a sequence of prompt completion tasks [1]. Bidirectional context modeling in code infilling achieves a pass rate of 69.0% in the single-line task compared to the left-to-right context model's 48.2% [12]. Documentation generation without fine-tuning has an 18.27 BLEU score compared to the fine-tuned CodeBERT model's 19.06 BLEU score [12]. This underlines the transformative potential of data engineering automation.

VelTech University, India

Other challenges for LLMs in enterprise data engineering include maintaining code quality across thousands of data pipelines created by engineers with varying skills, the output code meeting security policy and data classification/sovereignty rules, as well as the output code meeting regulatory requirements. Finally, to ensure true semantic correctness, the generated artifacts must be auditable, meaning that they can be traced back to their originating specifications, the model version, and the approving engineers. The risk of extraction attacks [11], through which proprietary data can be extracted from LLM prompts, creates a wide gulf between the theoretical capability of LLMs demonstrated through academic benchmarks and real-world deployment, especially in regulated enterprises.

To address these challenges, we propose a unified seven-phase orchestration framework that combines LLM code generation with enterprise governance, including quality gates, enterprise guardrails to prevent security violations, and human oversight through confidence-gated review. It also uses reinforcement learning to refine the LLM-generated code based on feedback from production. The design leverages findings: multi-turn synthesis reduces the perplexity of prompts and improves their pass rate [1], bidirectional context modeling improves infilling [12], and instruction tuning using human feedback considerably improves model alignment [9]. This enables our paper to provide an end-to-end architecture that takes into account research advances and enterprise deployment constraints.

2. Materials And Methods

2.1 Framework Architecture Overview

The framework adopts a sequential seven-stage quality-gated structure that includes the various challenges identified in the analysis of the LLM code generation research landscape as well as enterprise deployments. Stage boundaries check validation points, including the quality, security, and compliance of the generated artifacts prior to allowing them to begin the next stage. The architecture supports greenfield development from natural language specifications and migration translation from legacy code artifacts. Both pathways converge at several validation steps in terms of governance and orchestration.

To address these issues, key design principles of the system are decoupling generation from validation, encoding organizational knowledge in the system (retrieval-augmented generation), multi-level defense against security threats, human-in-the-loop control of high-stakes tasks, and continually learning from production data. These counter key challenges of the system, namely that the character of the generated texts are highly dependent on the ability to prompt the LLM, security threats are correlated with how generation is done, and decay in production output quality over time.

2.2 Stage 1: Specification ingestion and context enrichment

The first stage is responsible for parsing specifications, classifying them, and improving contexts using retrieval-augmented generation. An input classifier analyzes the submitted specification or legacy code artifact and extracts the intent, complexity indicators, data domains, and technology. This will define subsequent pipelines and initial quality scores. Specifically, the RAG retrieval system queries an organizational knowledge base that includes existing pipeline implementations, best practices for architectures and technologies, policies for data classification, authorized libraries, and business rules in their respective domains.

Retrieval-augmented generation (RAG) uses both the pretrained language models' parametric knowledge as well as non-parametric knowledge from external sources, which considerably improves performance on knowledge-intensive tasks [15]. The knowledge base consists of semantic embeddings, which are used to find relevant organizational context based on similarity matching. Retrieval retrieves the 15 most relevant patterns, standards, and rules from the embedding database using similarity scores with the input. The prompt assembler combines the input specification with the retrieved patterns, standards, and rules; relevant data classification; and target platform specifications into a structured prompt payload.

First, enrichment solves the problem that pass problems have lower prompt perplexity, the cross-entropy loss of the language model given the prompt, than non-pass problems (3.12 vs. 3.40 perplexity for 16.1B models) [1]. Improving the specifications with organizational context resolves this by adding clarity and including examples of implementations that conform to the specifications.

The enrichment imposes constraints post facto, preventing plaintext from being logged, requiring encryption protocols, and forcing field masking in non-production environments.

Component	Function	Key Techniques	Output
Input Classifier	Parses and classifies specifications	Intent extraction, complexity detection	Structured specification metadata
RAG Retrieval System	Fetches organizational knowledge	Semantic embeddings, similarity matching	Relevant patterns, policies, rules
Prompt Assembler	Combines inputs into structured prompts	Context injection, constraint encoding	Enriched prompt payload
Knowledge Base	Stores organizational intelligence	Embeddings, best practices	Context for generation
Constraint Enforcement	Applies compliance requirements	Data masking, encryption rules	Secure prompt constraints

Table 1. Stage 1 – Specification Ingestion and Context Enrichment Overview [1, 15]

2.3 Stage 2: LLM-Powered Multi-Stage Code Generation

The second stage performs LLM-powered code generation in multiple sub-stages with several stages of validation. For greenfield development, the stages are architecture generation, implementation synthesis, error handling, monitoring instrumentation, and test scaffolding. In migration scenarios, semantic parsing extracts platform-independent business logic from legacy code, providing idiomatic implementations for the target platform.

In building architectures, chain-of-thought (CoT) prompting is a technique used to improve reasoning capability. CoT prompting considerably improves performance on the difficult reasoning tasks requiring multiple steps of inference that standard prompting techniques fail at [17]. The output of this stage is a high-level architecture blueprint of the pipeline with respect to technology stack, resource allocation, data flow topology, and deployment configuration validated semantically against the extracted requirements.

This stage applies few-shot learning to the generated implementations and retrieves the examples from the organizational pattern library created in stage 1. Constrained generation enforces the organization patterns such as naming, code structures, and documentation formats. Multi-turn synthesis, which breaks complex pipelines down to a series of generation steps, reduces its prompt perplexity from 10.25 in a single turn to 8.05 for 16.1B models. Its pass rate exceeds that of single-turn concatenated specifications at 47.34% vs. 38.74% [1]. Each synthesis sub-stage then validates its output to prevent errors from propagating through the pipeline.

Using a semantic parser with code infilling methods, where both the prior and the following context is masked out, achieves a 69.0% acceptable pass rate compared to a 48.2% pass rate for left-to-right decoding and a 54.9% pass rate for a left-to-right reranking approach [12]. It extracts business logic, data transformations, dependencies, and quality constraints from the code, emitting an intermediate platform-independent representation. Target platform generation synthesizes idiomatic implementations with platform-specific optimizations, ensuring it retains the previously-extracted business logic. A verifier LLM compares semantic representations of source and target code and detects potential preservation issues for human review.

2.4 Stage 3: Automatic Documentation Generation

In the third stage, the LLM generates documentation immediately after code generation while the LLM context is warm. The following artifacts can be created: business logic descriptions data lineage narratives input-output schema documents ,

operational runbooks, and 'troubleshooting guides.' Business logic describes the purpose of the pipeline and why it is created and run. Data lineage describes how data is transformed in a pipeline. Input-output schema documents the names and types of fields. Operational runbooks describe deployment and monitoring procedures. Troubleshooting guides describe failure modes and how to diagnose and correct problems.

Zero-shot synthesis conditioning on generation of code and context is used for code docstring generation. Causal masked infilling with no fine-tuning achieves 18.27 BLEU scores on docstring tasks compared to 16.05 and 17.14 scores for left-to-right single and left-to-right reranking models, respectively [12]. These zero-shot results are comparable to the supervised baselines of CodeBERT, which achieves 19.06 BLEU with 250,000 examples, and CodeT5, which achieves 20.36 BLEU with 250,000 examples [12]. It also generates documentation without requiring supervised datasets, making it simple to use.

The generated documentation is committed to the organization's knowledge base, is used to improve the RAG corpus to ease further specification generation in Stage 1, creating a virtuous cycle of improvement, and is also put into a code search index so that semantic search can be performed across the organization's pipelines. Documentation generation addresses documentation lagging behind implementation, knowledge silos and troubleshooting. It does so by synthesizing documentation during its generation with the available context, enabling the framework to produce accurate and consistent documentation.

2.5 Stage 4: Multi-Layer Automated Validation

The fourth stage runs four parallel validation pipelines, which validate syntax, security, semantics, and performance. The syntax validation checks for correct syntax, code style, naming, and structural conventions and uses linters for a given programming language. Security scanners include SAST tools like SonarQube and Semgrep to find code vulnerabilities, hard-coded passwords SQL injection, insecure API usage, and policy violations. The security layer manages up-to-date vulnerability databases and organizational security policies and scans generated software against them.

Semantic validation uses another verifier LLM, which can check whether the logic follows the specifications, since syntactically valid code may not implement the specifications. The verifier performs requirement-to-implementation tracing and is able to identify missing requirements, incorrect implementations, and logic errors. In migration scenarios, semantic validation compares extracted source semantics with generated target semantics to verify business logic preservation.

The performance validation estimates resource consumption, execution duration, data volume, and cost against SLA thresholds. The estimator employs heuristics driven by execution of past pipelines and queries to analyze data scans, transforms, and generated volumes. Estimates of those metrics versus the SLAs help anticipate performance problems before deployment.

The result of validation is presented as a composite quality score. This score is a weighted aggregation of the results of validating the syntax (20%), security (35%), semantics (30%), and performance (15%); security and semantics are most important. The composite score (0 to 1) is used for tier routing in Stage 5. Validation implements defense in depth, checking generated code against multiple quality dimensions before it is reviewed by a human.

Validation Layer	Purpose	Tools/Approach	Weight (%)
Syntax Validation	Ensures code correctness and style	Linters, formatting tools	20
Security Validation	Detects vulnerabilities	SAST tools, policy checks	35
Semantic Validation	Verifies logic correctness	Verifier LLM, requirement tracing	30
Performance Validation	Estimates execution efficiency	Heuristics, historical analysis	15
Composite Score	Aggregated quality metric	Weighted scoring model	100

Table 2. Multi-Layer Validation Framework and Weight Distribution [1, 12]

2.6 Stage 5: Confidence-Gated Human Review

The final stage is tiered human review based on composite quality scores from Stage 4. The framework defines the five review tiers and their associated reviewers, actions, and SLAs. Automated deployments are made for artifacts scoring greater than 0.95 with spot-check times of less than 30 minutes. For artifacts scoring between 0.80 and 0.94, pull requests are made to assigned engineers within four hours. For all artifacts scoring between 0.65 and 0.79, a senior engineer performs a review, and regeneration options are discussed within eight hours. Between 0.50 and 0.64, the artifact has to be regenerated using a different specification and reviewed by the lead engineer within 24 hours. Items with scores below 0.50 are disapproved and referred to engineering management for specification rewrite.

This tiered approach is a tradeoff between the advantages of automation and the advantages of risk reduction by allowing humans to inspect the artifacts when confidence falls below some minimum threshold.

Migration artifacts receive the standard review tier regardless of quality score, as they are more likely to contain silent logic errors.

Review interfaces include diagnostics artifacts for original specifications, validation output, security scans including severity classification, and performance estimates including service level agreement threshold comparisons on all interfaces. Furthermore, the attribution generates language-level provenance traces over each generated code block, so the reviewer can see which input specification parts are responsible for generating each code portion to check for requirement satisfactions. The engineer can then accept, edit with inline commands, or reject with regeneration requests, and their choices get written to the RLHF learning store for reuse.

The confidence-gated approach addresses the need to increase production when human review is a bottleneck without risking low-quality outputs when the language model is uncertain about which generations it is creating. Quality signals enable increased productivity over human review for high-quality generations while restricting human review for low-confidence generations.

2.7 Stage 6: Deployment And Production Monitoring

During the sixth step, the approved artifacts are deployed using CI/CD tools that execute integration testing as a safeguard before being promoted to production. During this process, code is written to version control, builds are created, unit tests are executed (to test the working being done within an individual component), and integration tests are executed (pipeline execution tests performed using representative data from a staging environment). Integration tests verify that data flows and transforms through components properly, propagates any errors, and is instrumented for monitoring.

Once validated in the staging environment, they are deployed to production environments. Pipelines then include features for tracking execution performance, validating data quality and error rates, and tracking SLA achievements. Monitoring agents like CloudWatch or Datadog can capture metrics, logs, and traces that can be queried using statistical anomaly detection techniques that allow indications of performance degradation, data quality issues or error rate increases.

If any of the failures can be assigned to LLM code, the learning store will provide feedback signals as large negative rewards. Any artifact that a failure can be traced back to will be re-reviewed by a human before being re-released. This creates a closed loop feedback mechanism over the generation output that avoids regressions, even in systematic mode when production output is used. It also allows to gather empirical evidence on the quality of generated code under real workloads with real data and complexity.

2.8 Stage 7: Reinforcement Learning and Continuous Improvement

Stage 7 is online learning with reinforcement learning from human feedback (RLHF). When the human collects feedback on a Stage 5 decision, test results from Stage 6, security findings from Stage 4, and production incidents are aggregated, the reasoning is written to the RLHF learning store. The store acts as a memory of the outcome of generations concerning quality.

The RLHF training pipeline uses the feedback signals to calculate reward signals to further fine-tune the model. Artifacts whose quality is accepted are assigned positive rewards. Rejected artifacts receive a penalty proportional with the severity of the rejection. Production incidents are assigned a high penalty. For security issues, a heavily negative

reward based on severity classification is used, with the reward model further refined by standard RLHF techniques to improve the quality of future generations [9].

Accepted artifacts with a score greater than 0.90 are stored in the few-shot example repository as retrieval context for Stage 1. Documentation pattern examples with high scores are stored in the documentation pattern repository. Rejected patterns disallow bad code shape. Fine-tuning runs weekly; prior to release, new models run through an A/B shadow evaluation against production baselines, assessing both behavioral and performance regressions. A shadow evaluation will evaluate both the current and candidate model in production and compare their quality scores without deploying the candidate and may prefer candidates that achieve statistical importance.

This architecture for continuous improvement addresses the problem that static models decay relative to changing business objectives, hardware and software technologies, and data types. Production learning is a repeated process leading to improved model accuracy. For instance, instruction tuning with human feedback generally outperforms the base models in terms of helpfulness and harmlessness performance scores [9], which validates the continual learning approach.

2.9 Enterprise Guardrails and Compliance Controls

The framework supports end-to-end compliance, with completely automated guardrails and automatic data classification by imposing constraints on the generation prompt, depending on the data domain of the target pipeline. The prompt assembler includes a classification policy engine that maps the data domains of a pipeline to sets of constraints. Pipelines containing PII, PCI, or regulatory-sensitive data must not log sensitive fields, must encrypt data at rest with AES-256, must use TLS 1.3 for data in transit, and must mask sensitive fields in test environments. These constraints are injected into the generation context for inclusion in Stage 4 semantic verification.

The RAG retrieval system is limited to returning only metadata, including schema signatures, anonymized code snippets, documentation summaries, and filtered records. This guards against extraction attacks, where LLMs can sometimes be induced to leak training data. Whether this guards against extraction attacks in practice depends on architecture and training method [11]. Because training data extraction attacks can recover verbatim snippets from production models [11], policies should prevent proprietary data from entering LLM contexts.

Generation scope limits maintain a blocklist of code patterns that are considered unsafe or vulnerable, including hardcoded passwords, dynamic SQL statements that allow injection attacks, insecure deserialization, and unsafe file operations. The list is periodically updated. Stage 4 security scanning enforces these constraints by stripping these patterns from generated output. Infrastructure as code generation does not apply to security groups, IAM policies, or network configuration; these rules must be written by engineers with security expertise.

The orchestration writes append-only audit logs with immutable timestamps and data such as artifact checksums, specification hashes, context retrieval records, generation model version, validation results, reviewer identities, approval flags, deployments and monitors. Audit logs allow regulatory review to trace a production pipeline artifact generated by a model version to its original specification, validation results, and approving engineers. These capabilities can be useful in financial services or healthcare, where provenance of code is mandated by regulation.

3. Framework Design Rationale

3.1 Multi-Turn Synthesis for Complex Specifications

The framework utilizes a multi-turn synthesis mechanism by decomposing its specifications, resulting in substantially better generation. CodeGen has a 47.34% multi-turn pass rate for the 16.1B parameter models, compared to 38.74% for their variants using single-turn concatenated specifications [1]. In smaller models, pass rates improve considerably when compared to single-turn pass rates: 43.52% compared to 28.48% for 6.1B models and 38.72% compared to 25.43% for 2.7B models [1].

For example, in the case of the 16.1B model, multi-turn synthesis reduced prompt perplexity from 10.25 to 8.05. In the case of the 6.1B model, perplexity is reduced from 10.58 to 8.18, and for the 2.7B model, perplexity is reduced from 11.67 to

8.90 [1]. This shows improved understanding of specification and greater synthesis success. Analysis of the difficulty of problems indicates multi-turn factorization is helpful, as multi-turn pass rates for medium and hard problems in the 6.1B models are considerably higher than for single-turn problems [1]. These results motivated the creation of a multi-stage generation process in Stage 2, where the complex pipeline of architecture, implementation, error recovery monitoring, and testing are broken into a series of prompts instead of being a single-stage generation.

Additionally, since the larger models profited less from multi-turn on simple problems (0.19% for the 6.1B models and -0.25% for the 16.1B models), multi-turn synthesis is only performed on-the-fly when the complexity indicators computed as part of the Stage 1 specification classification indicate multi-turn is required. However, for simple pipelines with simple transformations, multi-turn decomposition is avoided to prevent latency loss. Pipelines with complicated business logic, multiple data sources, or deep data transformations may require a multi-turn processing mode.

3.2 Bidirectional Context for Migration and Infilling

The motivation for bidirectional context modeling in the migration path comes from code infilling research where conditioning on both previous and subsequent context generally offers a meaningful performance increase. For InCoder causal masked infilling, the 69.0% pass rate and 56.3% exact match on single-line tasks have been shown to be much better than the 48.2% pass rate and 38.7% exact match from left-to-right single-only and the 54.9% pass rate and 44.1% exact match from left-to-right reranking [12].

Models benefit from larger right-side context. When most function lines (80%) are on the right side, causal masked infilling outperforms left-to-right baselines by 30 points [12]. This inspired our Stage 2 migration design. The migration performs semantic parsing to extract the end-to-end business logic, including downstream usage patterns, before injecting it into the generation process, providing rich bidirectional context. This also means target platform generation is based on the entire source pipeline rather than being top-down incremental.

On semantic return type prediction task, the accuracy of causal masked infilling is 58.1% on the CodeXGLUE benchmark, compared to 12% and 12.4% for left-to-right single and left-to-right reranking [12]. On TypeWriter OSS test datasets, zero-shot causal masked infilling achieves 59.2% precision, recall, and F1 scores, compared to 54.9%, 43.2%, and 48.3% precision, recall, and F1 scores of supervised TypeWriter models [12]. This also extends to the Stage 4 semantic verification experiment, where the verifier LLM is conditioned on both specifications and full generated implementations, rather than testing at intermediate stages during implementation generation, producing better tracing from requirements to implementations.

3.3 Quality-Gated Review Routing

Given the benchmark pass@1 rates from 12.76% for 350M models to 29.28% for 16.1B models [1], we expect variation between generations, and to a lesser degree, model sizes. This is reason for our confidence-gated review design where we route higher quality generations to automated deployment and more uncertain generations to manual review. The 0.95 auto-merge threshold matches well, and high acceptance rates for highest-quality generations imply that human review provides little value when quality signals are strong.

The five-tier review system enables intuitive human intervention with the most promising generations. Generations with acceptance rates from 0.80 to 0.94 are reviewed in most cases, while generations with acceptance rates between 0.65 and 0.79 are reviewed if they could potentially be saved via further refinements during targeted rounds. Regeneration is mandatory below 0.64 to ensure low-quality code won't reach production even with human approval.

The composite quality score weighs security compliance and semantic correctness at 35% and 30%, respectively. This is aligned with the risks to enterprises, where either a breach or logic bug has higher cost implications than syntax or performance violations, which can be more easily remediated using tools or refactorings. Syntax correctness was scored at 20%, and performance efficiency at 15%.

3.4 Continuous Learning with RLHF

The Stage 7 continuous learning design implements RLHF. Instruction tuning with human feedback has been shown to improve performance on helpfulness and harmlessness evaluations compared to base models [9]. Tuning language models through direct

human feedback encourages the model to generalize to new instructions and to act safely and reduce harmful output [9]. Instruction tuning methods yield reliable performance increases for models across different tasks [10].

Feedback can come from human review decisions, test results, security findings, or production issues. Such informative multi-dimensional feedback can provide better signal to the training algorithms beyond only functional correctness feedback to the model. Weekly fine-tuning followed by A/B shadow evaluation before promotion is the best trade-off between improving the model and preventing regression. This shadow evaluation branch, running the current production model alongside the candidate model, evaluates the latter's performance without deploying it into the production environment.

Further, thanks to the inclusion of high-quality artifacts already accepted by the organization, the effect is cumulative: as more successful pipelines are accepted, the organizational context retrieved during Stage 1's retrieval-augmented generation process improves. This tackles the cold-start challenge of the authoring system's lack of organizational examples at each generation, strengthening the corpus of validated implementations.

Design Principle	Description	Supporting Evidence
Multi-Turn Synthesis	Breaks complex tasks into smaller steps	Higher pass rates and lower perplexity
Bidirectional Context Modeling	Uses full context in migration	Improved accuracy in code infilling
Quality-Gated Review Routing	Routes based on confidence scores	Balances automation and risk
Continuous Learning (RLHF)	Improves model via feedback loops	Better helpfulness and safety performance
RAG-Based Enrichment	Injects organizational knowledge	Enhances context and generation accuracy

Table 3. Key Design Rationale and Supporting Evidence [1, 9, 10, 12]

4. Discussion

We present a seven-stage orchestration framework that bridges the performance gap between LLMs on academic benchmarks and enterprise performance requirements in regulated domains. This end-to-end orchestration framework further integrates quality gates, enterprise guardrails, human-in-the-loop checks, and continual learning at each step of the orchestration process, following a phased approach. It captures architectures and empirical studies from recent code generation literature.

For Stage 2 multi-turn synthesis, we assume as our observation that 47.34% of multi-turn prompts pass vs 38.74% of single-turn prompts, and the average perplexity of multi-turn prompts is lower than that of single-turn prompts (10.25 vs 8.05) [1]. We utilize this understanding to adaptively switch between the different strategies based on the difficulty of the pipelines, since multi-turn synthesis has no clear benefit for easy problems with large models [1].

In Stage 2, bidirectional context modeling for migration is based on results achieving 69.0% and 48.2% pass rate in code infilling in comparison to left-to-right models. In semantic parsing, the business logic from the source is extracted before the code is generated for the target platform, resulting in rich bidirectional context. The dedicated verifier LLM allows examination of the fidelity of business logic preservation in the semantic-space unlike approaches where only syntactic translation is employed.

Stage 5's confidence-gated review routing uses composite quality scores from syntax, security, semantic, and performance validation to balance risk and automation, relying on human review only for uncertain generations while allowing confident high-quality (high-confidence) generations to be deployed autonomously. This addresses the

challenge that blanket automation introduces unacceptable risk while blanket human review eliminates the productivity benefits that enable LLM adoption.

Enterprise guardrails, such as data classification enforcement, metadata-only retrieval, generation scope limits, and immutable audit logs, show how responsible LLMs can be safely operated in regulated environments. The metadata retrieval policy counteracts data-extraction attacks that recover verbatim training sequences from production models [11]. A full audit trail links artifacts to specifications, models, results of validation, and approving engineers, to show compliance with provenance requirements in industries such as financial services and healthcare.

In Stage 7, iterative reinforcement learning from human feedback improves the quality of the model over time. Data from human comparisons, test results, security reviews, and real-life issues the model has encountered in production help the framework improve the model's performance. Once accepted, few-shot libraries benefit from the organizational context available during retrieval-augmented generation of new specifications leveraging existing quality artifacts.

There are some limitations to this work, as this architecture design is a proposal that requires production validation. While the architectural decisions have been made based on existing literature that shows the efficacy of multi-turn synthesis, bidirectional context modeling, and human feedback in improving generation quality, the entire seven-stage orchestration architecture has not been validated in production. The framework assumes access to organizational knowledge bases, fine-tuning infrastructure and monitoring tools, which may not be uniformly available in all organizations. It requires upfront investment for knowledge base development, validation pipeline design, and online fine-tuning infrastructure.

As this framework is focused on code generation, other aspects of data engineering such as data quality monitoring, schema evolution management, or operational incident response fall out of scope. The thresholds for the quality-gated routes should be based on benchmark acceptance rates, which are to be calibrated to the risk appetite and quality bar of each organization. The composite quality score weighting for security and semantic accuracy reflects general enterprise priorities but may be shifted for organizations with different risk profiles.

The next step is to apply this approach to production banking data platforms and measure the developer throughput, code quality, security outcomes and operational efficiency. We will measure how far the orchestration architecture achieves the productivity improvements that the capabilities of the individual components promise, and whether those components achieve enterprise quality and security. Future development will investigate multi-agent collaboration on pipelines, formal verification of mission-critical regulatory workloads, and infrastructure-as-code generation as LLM capabilities expand.

Aspect	Framework Approach	Supporting Evidence	Implication
Multi-Turn Synthesis	Decomposes complex specifications into sequential prompts	The pass rate improves from 38.74% to 47.34%; perplexity reduces from 10.25 to 8.05	Improves generation accuracy for complex pipelines
Bidirectional Context Modeling	Uses full source and target context during migration	Pass rate: 69.0% vs 48.2% in left-to-right models	Ensures better semantic preservation
Confidence-Gated Review	Routes artifacts based on composite quality score	Quality thresholds (0.95, 0.80, 0.65, 0.50)	Balances automation with risk control
Multi-Layer Validation	Combines syntax, security, semantic, and performance checks	Weighted scoring (35%, 30%, 20%, 15%)	Ensures enterprise-grade quality assurance
Enterprise Guardrails	Enforces data classification and secure generation constraints	Protection against extraction attacks	Enables safe deployment in regulated environments

Continuous Learning (RLHF)	Uses feedback from production and human review	Improved alignment via RLHF	Enhances model performance over time
Auditability and Compliance	Maintains immutable logs and traceability	End-to-end provenance tracking	Supports regulatory requirements

Table 4. Summary of Framework Contributions, Evidence, and Implications [1, 9, 11, 12]

Conclusion

To this end, we propose a seven-step orchestration framework that utilizes large language models within enterprise data engineering workflows, serving to bridge problems when moving from fine-tuned LLMs as a code generation tool to a productionized setting with quality gates, enterprise guardrails, human in-the-loop, and a continuous learning cycle. This work builds on recent findings in code generation research that multi-turn synthesis outperforms single-turn, that bidirectionally modeling the context is superior to left-to-right generation, and that instruction tuning with human feedback assists in aligning model behavior with organizational requirements.

The seven stages are specification ingestion, using RAG to infuse knowledge from the organization and engage with long prompts. Multi-stage code generation, decomposing complex pipelines into a series of code creation steps. Automatic documentation creation, producing documentation of a quality comparable to supervised models, without needing fine-tuning. Multi-layer validation, verifying syntax correctness, security compliance, semantic correctness, and runtime performance. Confidence-gated human review, selecting the most important tasks for human intervention while discarding lower quality outputs. CI/CD incorporating integration testing and production monitoring, ensuring consistency and reliability. Continuous learning, utilizing reinforcement feedback to gradually improve the generation process over time. Enterprise guardrails, including data classification enforcement, metadata-only retrieval policies, generation scope limits, and immutable audit trails.

The contribution is an architecture that delivers the needs of the LLM research ecosystem and the current state of LLM in enterprise. The architecture adds enterprise quality gates for generated artifacts, guardrails to prevent security lapses and data leaks, acceptable human involvement based on LLM generation confidence, and feedback loops to utilize production experience to improve generation. The proposed research is to validate the architecture through production deployment in banking data platforms. We will measure developer productivity, code quality, security vulnerability rate, and operations cost.

References

[1] Erik Nijkamp et al., "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," arXiv, 2023. DOI: https://doi.org/10.48550/arXiv.220313474 [Online]. Available: https://arxiv.org/abs/2203.13474

[2] Jagadeesan Srinivasan et al., "Detection and analysis of prompt injection in Indian multilingual large language models," Sci Rep (2026). https://doi.org/10.1038/s41598-026-43883-0 [Online]. Available: https://www.nature.com/articles/s41598-026-43883-0_reference.pdf

[3] Xing Xu et al., "Structure-Aware Lightweight Document-Level Event Extraction via Code-Based Large Language Models," Electronics 2026, 15(6), 1187; https://doi.org/10.3390/electronics15061187 [Online]. Available: https://www.mdpi.com/2079-9292/15/6/1187

[4] Leonardo Ranaldi et al., "Multilingual Retrieval-Augmented Generation for Knowledge-Intensive Question Answering Task," Findings of the Association for Computational Linguistics: EACL 2026, pages 697–716, March 24-29, 2026. [Online]. Available: https://aclanthology.org/2026.findings-eacl.35.pdf

[5] Haodong Chen et al., "Beyond GeneGPT: A Multi-Agent Architecture with Open-Source

LLMs for Enhanced Genomic Question Answering," Proceedings of the 2025 Annual International ACM SIGIR Conference on Research and Development in Information Retrieval in the Asia Pacific Region (December 2025) https://doi.org/10.1145/3767695.3769488 [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3767695.3769488

[6] Shunyu Yao et al., "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," 37th Conference on Neural Information Processing Systems (NeurIPS 2023). [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf

[7] Federico Cassano et al., "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation," "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," IEEE Transactions on Software Engineering 49.7 (2023): 3675-3691. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=10103177

[8] Ning Miao et al., "Self-check: Using LLMs to zero-shot check their own step-by-step reasoning." arXiv preprint arXiv:2308.00436 (2023). [Online]. Available: https://arxiv.org/pdf/2308.00436

[9] Long Ouyang et al., "Training language models to follow instructions with human feedback," Advances in neural information processing systems 35 (2022): 27730-27744. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf

[10] Shengyu Zhang et al., "Instruction tuning for large language models: A survey." ACM Computing Surveys 58.7 (2026): 1-36. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3777411

[11] Nicholas Carlini et al., "Extracting training data from large language models," 30th USENIX security symposium (USENIX Security 21). 2021. [Online]. Available: https://www.usenix.org/system/files/sec21-carlini-extracting.pdf

[12] Daniel Fried et al., "Incoder: A generative model for code infilling and synthesis," arXiv preprint arXiv:2204.05999 (2022). [Online]. Available: https://arxiv.org/pdf/2204.05999

[13] Ahmed Soliman et al., "Leveraging pre-trained language models for code generation," Complex & Intelligent Systems 10.3 (2024): 3955-3980. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s40747-024-01373-8.pdf

[14] Sebastian Eggers, "Automating Data Lineage and Pipeline Extraction," Proceedings of the VLDB Endowment. ISSN 2150 (2024): 8097. [Online]. Available: https://www.vldb.org/2024/files/phd-workshop-papers/vldb_phd_workshop_paper_id_11.pdf

[15] Yucheng Hu and Yuxing Lu , "Rag and rau: A survey on retrieval-augmented language models in natural language processing," arXiv preprint arXiv:2404.19543 (2024). [Online]. Available: https://arxiv.org/pdf/2404.19543?

[16] B. Roziere et al., "A systematic evaluation of large language models of code." Proceedings of the 6th ACM SIGPLAN international symposium on machine programming. 2022. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3520312.3534862?utm_source=consensus

[17] Jason Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," Advances in neural information processing systems 35 (2022): 24824-24837. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf