

# The Next Interface: Humans and AI Co-Create End-to-End Software

Advancing Low-Code/No-Code Pipelines with Multi-Agent Systems and Large Language Models

---

## ARTICLE INFO

### Keywords:

Large Language Models (LLMs)  
Generative AI  
Natural Language Processing (NLP)  
Automated Software Generation  
Human-AI Co-creation  
Low-Code/No-Code (LCNC)  
Multi-Agent Systems (MAS)

## ABSTRACT

Large language models and foundation models are reshaping information processing by enabling natural-language inputs to be transformed into structured digital artifacts. In software engineering, however, direct LLM-based code generation often remains difficult to control, explain, and validate, while traditional low-code/no-code platforms provide reliability at the cost of flexibility. This study addresses that gap by proposing a multi-agent framework that integrates foundation-model capabilities with structured representations and automated DevOps processes to support end-to-end software generation from natural-language requirements. The framework introduces a semantically enriched YAML blueprint as an intermediate representation between user intent and code synthesis. This artifact enables coordinated interaction among five specialized agents responsible for requirements interpretation, template-based generation, business-logic adaptation, automated validation, and cloud deployment. From an information-processing perspective, the approach functions as a structured transformation pipeline that converts informal textual descriptions into formalized and executable software artifacts while improving traceability, transparency, and architectural consistency. Evaluated through a Design Science Research case study on a corporate income tax management application, the framework achieved 95% accuracy in entity-relationship mapping, 85% business-rule accuracy versus 50–60% for direct LLM generation, a 90% automated test pass rate, and a 60% reduction in development time, while reducing manual intervention to 20% and improving architectural consistency from low to high. Beyond automation, the study positions natural language as an emerging interface for human–AI co-creation. By allowing non-technical stakeholders to participate more directly in software design, the framework contributes to more inclusive and accessible forms of digital production while promoting more trustworthy, explainable, and governance-aware AI-supported development. Future research should explore adaptive agent orchestration, incorporating learning mechanisms that allow development pipelines to dynamically optimize task allocation, validation strategies, and generation workflows. Further investigation is needed into human-AI co-creation interfaces, governance mechanisms, and socio-technical implications to better understand how foundation-model-driven development environments may reshape digital innovation.

---

## 1. Introduction

Large language models (LLMs) have rapidly emerged as a central paradigm in artificial intelligence, enabling machines to process, interpret, and generate complex forms of human language at unprecedented scale. Their ability to transform natural language into structured artifacts, executable code, and domain-specific knowledge representations has attracted growing attention across information systems, software engineering, and digital transformation research (Ferrari and Spoletini, 2025; Hemmat et al., 2025; Chechik et al., 2026). In particular, recent advances in foundation models have made it possible to envision development environments in which natural-language interaction becomes a primary interface for designing and generating software systems.

This shift is especially significant for software development, where natural-language requirements traditionally must be translated through multiple layers of abstraction before becoming executable systems. Requirements engineering, architectural design, implementation, testing, and deployment are typically handled by different specialists and tools, making the development lifecycle both technically complex and organizationally fragmented. While LLMs have demonstrated promising capabilities for code generation and automated reasoning, direct prompt-based generation often suffers from architectural inconsistency, lack of traceability, and limited integration with established engineering workflows (Ferrari and Spoletini, 2025; Hemmat et al., 2025; Warnett et al., 2026). At the same time, Low-Code/No-Code (LCNC) platforms have improved accessibility to software development but remain constrained by rigid

templates and predefined abstractions that limit flexibility for complex or novel applications (Sodano, DeFranco and Laplante, 2025; Martinez-Lasaca, Diez, Guerra and de Lara, 2025).

These limitations highlight a broader challenge at the intersection of foundation models and information processing: how to transform unstructured natural-language intent into reliable, structured, and operational digital artifacts while maintaining transparency, governance, and architectural coherence. Recent research suggests that addressing this challenge requires more than improved language models alone. Instead, it calls for new forms of system architecture that combine the interpretive capabilities of LLMs with structured representations, workflow orchestration, and validation mechanisms capable of supporting dependable software generation (Ferrari and Spoleetini, 2025; Chechik et al., 2026; Chowa, Alvi, Rahman, Rahman, Raiaan, Islam, Hussain and Azam, 2026).

In response to this challenge, this paper proposes a multi-agent framework that integrates foundation-model capabilities with structured design representations and automated DevOps processes to enable end-to-end software generation from natural-language specifications. Rather than relying on direct LLM code synthesis, the proposed architecture introduces a semantically enriched intermediate representation that translates user requirements into a structured design artifact before code generation begins. This design artifact acts as a bridge between natural-language intent and executable systems, enabling coordinated interaction between specialized agents responsible for architecture design, template-based code generation, business logic integration, validation, and deployment.

From an information-processing perspective, the framework can be viewed as a structured transformation pipeline that progressively converts informal textual input into formalized system representations and operational software artifacts. Each stage of the pipeline introduces additional structure and verification, thereby reducing ambiguity and improving traceability across the development lifecycle. By distributing tasks across specialized agents, the system also reflects emerging paradigms in agentic AI, where complex problem-solving is achieved through coordinated collaboration among multiple intelligent components rather than monolithic model interaction (Chowa et al., 2026; Chechik et al., 2026).

The research is conducted using a Design Science Research methodology, focusing on the design and evaluation of a technological artifact intended to address practical problems while generating generalizable design knowledge (Hevner, March, Park and Ram, 2004; Peffers, Tuunanen, Rothenberger and Chatterjee, 2007; Hevner, 2024; Baskerville, Pries-Heje and Venable, 2026). The proposed framework is evaluated through a realistic case study involving the automated generation of a corporate income tax management application, a domain characterized by complex rules, multiple interacting entities, and strict validation requirements.

This study contributes to the emerging literature on foundation-model-enabled information systems in several ways. First, it proposes a structured architecture for integrating LLMs into software engineering workflows through intermediate representations that improve transparency and architectural coherence. Second, it advances the understanding of multi-agent collaboration in AI-assisted development environments by demonstrating how specialized agents can coordinate across the software lifecycle (Chowa et al., 2026; Chechik et al., 2026). Third, it provides practical insights into how natural-language-driven application generation can be embedded within DevSecOps pipelines to produce deployable and validated systems. Finally, it highlights the broader societal implications of AI-assisted software generation by showing how such systems may reduce barriers to participation in software creation and support more inclusive forms of digital innovation (Sodano et al., 2025; Ferrari and Spoleetini, 2025).

By combining large language models, semantically enriched design artifacts, and coordinated multi-agent workflows, the proposed framework illustrates how foundation models can move beyond isolated generation tasks toward integrated information-processing infrastructures capable of supporting reliable and scalable software development.

The remainder of this paper is structured as follows. Section 2 reviews related work in LCNC platforms, AI-driven software engineering, and multi-agent systems. Section 3 consolidates the identified gaps, formally presents our research objectives and proposes a way forward. Section 4 outlines our methodology. Section 5 details our proposed architecture. Section 6 presents a use case demonstrating the framework's applicability. Section 7 discusses the implications of our approach and future research directions.

## 2. Background and Related Work

The proposed framework integrates research from three critical domains: AI-assisted software engineering, low-code/no-code (LCNC) platforms, and automated DevOps. While significant progress has been made in each area independently, their combination within a cohesive, agent-based architecture for end-to-end application generation

from natural language represents a novel synthesis. This section surveys the relevant literature, highlighting both the foundations upon which we build and the specific gaps our work addresses.

## 2.1. From Natural Language to Code Generation

The translation of natural language into executable code has become one of the most visible applications of large language models (LLMs) in software and systems engineering. Recent work shows that LLMs are no longer limited to lightweight code completion, but are increasingly used to generate structured artifacts, executable logic, and domain-constrained design outputs from textual descriptions. This trend is especially apparent in domains where natural-language inputs must be mapped to formal representations under explicit constraints, such as requirements engineering, business process modeling, and engineering design Ferrari and Spoletini (2025); Hemmat et al. (2025); Zheng, Han, Chen, Cao, Lu and Lin (2026); Hörner, Schacht, Fill et al. (2026); Kourani et al. (2025).

In requirements engineering, recent studies emphasize both the promise and the limits of LLM-based natural-language processing for transforming informal requirements into more structured and analyzable specifications. Ferrari and Spoletini argue that the relationship between formal methods and LLMs should be understood as a two-way roadmap: formal methods can provide correctness-oriented discipline to LLM-based workflows, while LLMs can make formal techniques more accessible in practice Ferrari and Spoletini (2025). Likewise, Hemmat et al. identify opportunities for using LLMs to support requirements analysis, refinement, and ambiguity reduction, while also emphasizing the persistent challenges of trustworthiness, traceability, and methodological control Hemmat et al. (2025).

Comparable developments can be observed in adjacent domains where natural language must be transformed into operational or semi-formal artifacts. In construction and regulatory checking, Zheng et al. show that LLM-driven function matching and composition can translate regulatory clauses into executable code for building design checking, thereby demonstrating the feasibility of converting textual constraints into machine-actionable logic Zheng et al. (2026). In process and conceptual modeling, recent work shows that LLMs can support the generation of BPMN process models and the acquisition of structured process knowledge from natural-language descriptions, although model quality, evaluation rigor, and domain-specific guidance remain decisive factors in performance Hörner et al. (2026); Kourani et al. (2025); Schinckus, Simonofski and Bono Rosselló (2025). Related work in digital humanities also shows that LLMs can support structured annotation and named entity interpretation in complex textual corpora, while recent named-entity-recognition research highlights the broader movement toward more entity-aware and knowledge-sensitive language-processing frameworks Galka and Vogeler (2026); Wang, Xie, Hu, Liu, Yu and Li (2025).

At the same time, the literature also makes clear that natural-language-to-code generation remains highly sensitive to prompting strategy, contextual information, and domain constraints. Walczak et al. show that both code context and prompting strategy materially affect the quality of LLM-generated unit tests, underscoring the dependence of generation quality on surrounding information and interaction design Walczak, Tomalak and Laskowski (2026). More broadly, the current evidence suggests that prompt-based generation alone is rarely sufficient when robust, auditable, and reusable software artifacts are required. Instead, stronger intermediate representations, explicit constraint handling, and better integration with engineering workflows remain necessary to bridge the gap between natural-language intent and dependable software artifacts Ferrari and Spoletini (2025); Hemmat et al. (2025); Warnett et al. (2026); Chechik et al. (2026).

These findings are especially important for business-oriented software generation. While LLMs can increasingly interpret domain language and produce executable or semi-executable outputs, the literature consistently shows that successful generation depends on more than linguistic fluency alone. Reliability, architectural control, lifecycle integration, and verifiable intermediate structures remain central requirements if natural-language-driven generation is to move beyond isolated prototype outputs toward disciplined software engineering practice Ferrari and Spoletini (2025); Hemmat et al. (2025); Chechik et al. (2026).

## 2.2. AI-Powered Code Adaptation and Program Repair

Automated Program Repair (APR) has traditionally focused on modifying existing code to satisfy specifications, correct faults, or restore expected behavior. Classical APR methods relied primarily on search-based strategies, transformation rules, fault localization, and test-suite validation. These approaches were largely designed around the assumption that bugs could be localized within existing code structures and repaired through relatively constrained transformations. In recent years, however, the field has expanded significantly through the integration of large language

models (LLMs), which can support patch generation, debugging assistance, repair candidate synthesis, and context-sensitive adaptation workflows Dikici (2025); Zubair, Al-Hitmi and Catal (2025). This shift is important because it moves APR beyond narrowly bounded mutation-based repair and toward more flexible forms of inference that can exploit both code-level and natural-language information.

The emerging literature suggests that LLM-based repair goes beyond purely syntactic mutation by making richer use of contextual information, such as natural-language bug reports, repository context, and dynamically generated tests. For example, Zhang et al. show that LLM-generated test cases can facilitate APR workflows and improve repair effectiveness, indicating that LLMs can contribute not only to patch generation itself but also to the auxiliary artifacts that guide the repair process Zhang, Wang, Liu et al. (2026). At the same time, empirical studies show that repair performance remains highly sensitive to benchmark design, programming language, model configuration, and fault-localization assumptions Jin et al. (2024); Campos et al. (2025); Zhang et al. (2026). These findings suggest that, although LLM-based repair improves flexibility and contextual reasoning, its generalizability across heterogeneous systems remains limited. They also indicate that performance gains often depend heavily on experimental setup, which complicates claims about robustness in realistic development environments.

More broadly, most existing approaches still treat AI as a reactive debugging tool applied after defects appear, rather than as part of a proactive software-generation architecture Dikici (2025); Zubair et al. (2025). This limitation becomes even more visible when APR is considered alongside recent work in requirements engineering and model-based software engineering, which shows that natural-language-driven generation remains unreliable unless it is supported by stronger formalization, intermediate representations, and architectural discipline Ferrari and Spoletini (2025); Hemmat et al. (2025); Chechik et al. (2026). Related work in process modeling and regulatory-code translation reaches a similar conclusion: transforming informal textual descriptions into operational software artifacts requires explicit quality control, domain grounding, and structured transformation steps Schinckus et al. (2025); Hörner et al. (2026); Kourani et al. (2025). Taken together, these adjacent strands of research suggest that the core challenge is not only how to repair faulty code once generated, but also how to embed repair within a broader pipeline that constrains, interprets, and validates software artifacts throughout development.

From this perspective, program repair should be understood not only as a downstream correction mechanism, but also as one component of a broader controlled adaptation process embedded throughout the software development lifecycle. Recent discussions therefore suggest that AI-powered repair should increasingly be treated as a modular capability within larger development workflows that also include requirements interpretation, test generation, intermediate modeling, and lifecycle-level validation. This view aligns with emerging work on agentic and workflow-oriented software engineering, which advocates more structured, auditable, and architecture-aware uses of AI in development environments Warnett et al. (2026); Chechik et al. (2026).

### 2.3. Multi-Agent Systems for Complex Workflow Orchestration

The recent rise of multi-agent systems built on large language models reflects a broader shift from single-prompt generation toward orchestrated workflows involving decomposition, specialization, tool use, and iterative coordination. Rather than relying on a single model to solve an entire problem in one step, multi-agent approaches distribute subtasks across role-specialized agents that interact through structured exchanges and intermediate outputs. Recent survey work characterizes this development as part of the emergence of agentic AI, in which LLMs are increasingly embedded in systems capable of planning, collaboration, memory use, and action selection across multi-step tasks Chowa et al. (2026); Chechik et al. (2026).

The appeal of this paradigm is especially clear in structured domains where tasks require multiple forms of expertise or sequential reasoning. Schinckus et al. show that a multi-agent approach can support process knowledge acquisition by organizing elicitation and formalization activities into structured stages linked to BPMN-oriented outcomes Schinckus et al. (2025). In engineering design, Chen and Bao present a multi-agent LLM framework for code-compliant automated design of reinforced concrete structures, showing how specialized agents can coordinate subtasks, cross-check outputs, and improve transparency in domain-constrained workflows Chen and Bao (2025). Similarly, Pan et al. demonstrate that an LLM-enabled multi-agent architecture for graph-based digital twins can ground natural-language interaction in structured graph representations, improving robustness and reducing hallucination relative to less structured baselines Pan, Zhang, Li, Wang, Liu and Chen (2026).

Taken together, these studies suggest that the strength of multi-agent LLM systems lies not merely in parallelizing work, but in structuring cognition through decomposition, grounding, verification, and controlled interaction. In this respect, multi-agent architectures are particularly relevant for software and systems engineering contexts where reliable

transformation from natural language to executable or semi-formal artifacts requires intermediate checking and explicit coordination across tasks Schinckus et al. (2025); Chen and Bao (2025); Pan et al. (2026). The broader literature on process modeling, requirements engineering, and model-based software engineering reinforces this point by showing that dependable generation depends on stronger workflow structure than prompt-based interaction alone can provide Ferrari and Spoletini (2025); Hemmat et al. (2025); Chechik et al. (2026).

Nevertheless, important limitations remain. Recent reviews of agentic LLM systems emphasize continuing challenges in evaluation, robustness, verifiability, and real-world coordination, especially when autonomous agents must operate across complex workflows with accountability requirements Chowa et al. (2026). This is particularly relevant in software engineering, where generated outputs must align with requirements, architecture, validation practices, and deployment constraints. Warnett et al. show that low-code pipeline generation still benefits from explicit structuring and lifecycle integration, while recent discussions in model-based and agentic software engineering argue for more architecture-aware forms of AI support rather than loosely coupled collections of agents Warnett et al. (2026); Chechik et al. (2026).

These observations imply that the future value of multi-agent systems in software development will depend less on the mere presence of multiple agents and more on how well agent roles are aligned with engineering stages, intermediate artifacts, and governance mechanisms. For complex workflow orchestration, the key challenge is therefore not only to make agents collaborate, but to ensure that collaboration is auditable, domain-grounded, and tightly integrated with the broader software lifecycle Chowa et al. (2026); Chechik et al. (2026); Ferrari and Spoletini (2025).

### 3. Identified Gaps and the Way Forward

Although commercial low-code development platforms (LCDPs) have significantly lowered the barrier to application development, they remain constrained by fundamental architectural limitations that restrict flexibility, scalability, and intelligent automation. Academic approaches, meanwhile, often address narrower technical challenges without fully resolving the broader problem of end-to-end trustworthy software generation. This section synthesizes the principal limitations identified across both research and practice and introduces the pathway proposed in this work.

#### 3.1. Critical Conceptual Gaps

A closer examination of the current landscape reveals several conceptual limitations that hinder the development of reliable and fully automated software generation systems. Despite the rapid progress in both AI-driven code generation and Low-Code/No-Code platforms, important structural and methodological gaps remain unresolved. The following key challenges summarize the most significant barriers identified in the literature and motivate the need for a more integrated and robust approach to automated software development.

**1. The Flexibility-Reliability Divide:** A fundamental tension exists between the flexibility offered by modern LLMs and the reliability provided by traditional development approaches. LLMs demonstrate unprecedented capability in generating code from natural language but suffer from non-determinism, architectural inconsistency, and inability to enforce governance standards Liu, Guo, Zhang, Ma, Li and Liu (2025a). Conversely, LCNC platforms and template-based systems provide structural reliability but are inherently bounded by their pre-built components and templates, limiting their applicability for novel or highly customized applications. This dichotomy represents a critical barrier to achieving truly adaptive yet trustworthy automated software generation.

**2. The Intermediate Representation Void:** Current approaches lack a robust intermediate representation that can bridge the semantic gap between natural language intent and executable code. While model-driven engineering approaches like Dandelion+ Martinez-Lasaca et al. (2025) offer formal modeling capabilities, they require explicit human specification and lack the interpretive power to derive models from natural language. Direct LLM code generation bypasses this crucial step, leading to the architectural coherence challenges identified in empirical studies Liu et al. (2025a). The absence of a verifiable, semantically rich intermediate representation prevents systematic validation and deterministic transformation throughout the development lifecycle.

**3. Business Logic Integration Challenge:** Existing systems struggle with formal representation and integration of complex business rules. Commercial LCNC platforms typically handle business logic through visual workflows or custom code escapes, while template-based generators provide skeletal structures without domain-specific behavior. There is no standardized methodology for representing business processes in a way that enables automated, reliable integration into generated codebases, particularly for domain-specific constraints and processes beyond basic CRUD functionality.



**4. Fragmented Automation Pipeline:** Current solutions address isolated aspects of software development but fail to provide end-to-end automation. LLMs generate code snippets but lack deployment integration; LCNC platforms facilitate rapid prototyping but require manual intervention for sophisticated validation and deployment; template generators provide starting points but leave business logic implementation to developers. This fragmentation reintroduces bottlenecks and prevents the realization of truly autonomous software development.

**5. Limited Personalization and Adaptation:** Both commercial and academic approaches offer limited capacity for personalization based on user profiles, technical requirements, or organizational contexts. Template-based systems provide static outputs, while LCNC platforms operate within vendor-defined boundaries. The absence of adaptive generation that considers user expertise levels, company technical standards, and specific skill requirements limits the practical applicability of automated systems across diverse user populations.

### 3.2. Limitations of existing platforms

An analysis of leading commercial LCDPs—including OutSystems OutSystems (2025), Mendix Mendix (2025), Zoho Creator Zoho Corporation (2025), Microsoft Power Apps Microsoft Corporation (2025), Google AppSheet Google LLC (2025), Kissflow Kissflow Inc. (2025), Salesforce Platform Salesforce, Inc. (2025), Appian Appian Corporation (2025), and Caspio Caspio, Inc. (2025)—reveals several consistent limitations:

**1. Fixed, Predefined Abstraction Layers:** These platforms operate within rigid, pre-configured abstraction boundaries. OutSystems OutSystems (2025) and Mendix Mendix (2025), for instance, provide visual editors for database schemas and UI components but lack mechanisms for defining fundamentally new domain-specific abstractions beyond their built-in paradigms. This constrains developers to solutions that fit within the platform's predetermined conceptual framework.

**2. Deterministic, Human-Orchestrated Workflows:** Behavior in these systems is typically implemented through visual workflows (e.g., Mendix Microflows, OutSystems server actions) that require explicit, manual orchestration of every step and decision point Mendix (2025); OutSystems (2025). While Zoho Creator Zoho Corporation (2025) and Microsoft Power Apps Microsoft Corporation (2025) offer event-driven automation, the logic remains entirely predetermined by human developers, lacking any capacity for adaptive reasoning or autonomous problem-solving.

**3. Limited Intelligent Assistance:** These platforms primarily assist with *implementation* through drag-and-drop interfaces and component libraries. They lack integrated agents capable of understanding high-level intent, generating novel solutions, or providing strategic architectural guidance. The "intelligence" is confined to pre-built templates and formula suggestions, not generative or reasoning capabilities.

**4. Vendor-Defined Development Trajectory:** The evolution path of applications is largely constrained by the platform's feature set. Extending beyond these boundaries typically requires resorting to custom code or complex integrations, defeating the purpose of a unified low-code environment. This creates the vendor lock-in problem that tools like Dandelion+ attempt to solve through model-driven engineering, but even such approaches still rely on human-driven model design and workflow specification.

Commercial and academic approaches to low-code development have made significant strides in democratizing software creation, yet they remain constrained by fundamental limitations in flexibility and automation.

**Commercial LCNC Platforms:** Low-Code/No-Code platforms such as Mendix, OutSystems, and Microsoft Power Apps have gained widespread adoption by democratizing software development through visual modeling and component-based assembly Zouani, Abdali and Ait Zaoui (2021). They significantly reduce the need for manual coding, enabling "citizen developers" to create applications rapidly Ihrwe, Di Ruscio, Mazzini, Fraternali and Pierantonio (2020). These platforms are inherently template-driven, relying on libraries of pre-built components to construct common application patterns, most notably CRUD (Create, Read, Update, Delete) interfaces Zouani and Lachgar (2024).

A well-documented limitation of these platforms is the *abstraction ceiling* Di Ruscio, Kolovos, de Lara, Pierantonio, Tisi and Wimmer (2022). While excellent for applications that fit their template library, they struggle with highly custom or novel functionality. Overcoming this often requires "escaping" to pro-code environments, which breaks the LCNC paradigm and reintroduces complexity. Similarly, traditional project scaffolding tools like Yeoman or Spring Initializr provide a static starting point but leave the intricate task of implementing business logic entirely to the developer.

**Academic and Model-Driven Approaches:** Beyond commercial platforms, academic research has explored model-driven engineering (MDE) approaches to low-code development. Frameworks like **Dandelion+** Martinez-Lasaca et al. (2025) represent significant advances by introducing explicit meta-modeling and workflow orchestration

through their PLATFLOW language. Dandelion+ addresses vendor lock-in concerns through technology-agnostic code generation and offers sophisticated access control mechanisms through constraint-based permissions.

However, Dandelion+ and similar MDE-based approaches still operate within a *human-orchestrated* paradigm. While they provide powerful abstractions for defining domain-specific LCDPs, the development process remains dependent on manual specification of meta-models, workflows, and access control policies. Our multi-agent architecture fundamentally shifts this paradigm by automating not just the code generation, but the entire *design and specification process* through natural language interpretation and AI-driven decision making.

### 3.3. Research Objectives

This research aims to address the fundamental limitations in current Low-Code/No-Code platforms and AI-driven code generation approaches by developing a comprehensive multi-agent framework for end-to-end software automation. The primary objectives of this study are:

#### 3.3.1. Structured Representation through Decorated Class Diagrams

To design and implement a transformation mechanism that converts natural language user input into a semantically decorated class diagram, serving as a structured intermediate representation that guides and frames subsequent code generation. This representation overcomes the ambiguity of direct LLM code generation by capturing application structure, business constraints, and semantic metadata before code synthesis begins.

#### 3.3.2. Architectural Coherence in AI-Generated Software

To leverage the decorated class diagram as a blueprint that ensures architectural consistency in AI-generated applications, overcoming the limitations of direct LLM code generation which often produces inconsistent or non-cohesive codebases. The semantic decorators (ROLE, STATE, ACTOR, MENU, STYLE, REF, REQ) provide explicit generation directives for security, interaction, and user experience layers.

#### 3.3.3. End-to-End Automation Pipeline

To develop a fully automated pipeline that spans the complete software development lifecycle—from natural language specification to cloud deployment—using the decorated class diagram as the central artifact that maintains consistency across all generation phases, addressing the current fragmentation in LCNC platforms.

#### 3.3.4. Business Logic Integration Framework

To create a formal methodology for representing and integrating complex business rules into generated applications through process triplets (initialization, validation, execution), enabling the framework to handle domain-specific constraints and processes beyond basic CRUD functionality, with the decorated diagram providing the structural context for logic adaptation.

#### 3.3.5. Multi-Agent Collaboration Mechanism

To establish an effective orchestration mechanism for specialized AI agents that collaboratively transform the decorated class diagram into production-ready software, ensuring deterministic outcomes while maintaining the semantic integrity captured in the original design representation.

#### 3.3.6. Validation and Quality Assurance

To implement an automated validation system that ensures generated applications meet enterprise-grade quality standards, using the decorated class diagram as a reference model for verifying functional correctness, security compliance, and architectural adherence.

The successful achievement of these objectives will be demonstrated through a comprehensive case study involving corporate income tax calculation, measuring improvements in development time, code quality, and automation completeness compared to traditional development approaches.

### 3.4. Our Integrated Approach Forward

Our architecture is designed in response to several persistent limitations in AI-assisted software engineering. A central issue in recent work is that direct LLM-based generation often produces outputs that are syntactically plausible yet architecturally unstable, insufficiently constrained, or difficult to verify across development stages. To address this, the Software Designer Agent does not generate code directly. Instead, it uses a fine-tuned LLM to produce a

structured, semantically decorated YAML blueprint that serves as an intermediate design artifact. Inspired by model-driven engineering principles (Brambilla, Cabot and Wimmer, 2017), this blueprint introduces an explicit abstraction layer between requirements interpretation and code synthesis. In doing so, it makes application structure, constraints, and semantic intent inspectable before implementation begins, thereby helping to mitigate the architectural coherence problems identified in empirical studies of AI-generated software. This design also builds on prior research in programmatic context understanding (Iyer, Konstas, Cheung and Zettlemoyer, 2021) and domain-sensitive language processing for specialized domains, including legal text analysis (Dehghani, Dehghani, Naderzadeh Ardebili and Rahnamayan, 2025), entity recognition (Zhang et al., 2026), and technology evolution modeling (Yang, Cai, Liu, Le, Zhang, Bissyandé, Liu and Tian, 2025).

A second limitation in the literature is that most AI-based code adaptation and repair approaches are oriented toward existing codebases, where the objective is to identify faults, synthesize patches, or restore expected behavior in mature systems. Our framework addresses a different problem. Rather than repairing legacy code, the Business Logic Adaptor Agent operates after structural code generation has taken place, enriching a scaffolded application with domain-specific business logic. This post-scaffolding intervention is important because it separates concerns between structural generation and behavioral specialization. The Template-Based Generator can therefore focus on producing a stable architectural foundation, while the Adaptor introduces more contextual process logic in a controlled manner. To support this, business processes are formalized through triplets of initialization, validation, and execution logic, and each process is encapsulated within a dedicated class. This modular organization improves controllability, keeps modifications localized, and reduces the likelihood that introducing one rule will unintentionally affect unrelated parts of the system. In that sense, the framework extends AI-assisted code transformation from reactive debugging toward proactive and structured logic integration within automated generation workflows.

These design choices collectively define the main contributions of the proposed framework. Specifically, the architecture combines interpretive flexibility, structural reliability, and lifecycle automation in a coordinated system.

Our research addresses the identified gaps through a multi-agent architecture that combines interpretive flexibility, structural reliability, and lifecycle automation in a single coordinated framework.

1. **Hybrid generation strategy for flexibility and control:** The framework combines LLM-based requirement interpretation with deterministic, template-based code generation. This hybrid strategy preserves the expressive power of natural-language interaction while anchoring outputs in predefined architectural patterns, helping reconcile adaptability with consistency.
2. **Semantically enriched intermediate design representation:** The framework introduces a YAML-based intermediate representation that captures application structure, business constraints, and semantic metadata before code synthesis begins. This design artifact supports verification, traceability, and design-level validation, while providing explicit directives for downstream generation.
3. **Structured integration of business logic:** The architecture formalizes business logic through process triplets comprising initialization, validation, and execution stages. This enables systematic incorporation of domain-specific behavior into generated applications and helps move beyond basic CRUD-oriented generation toward functionally richer systems.
4. **End-to-end multi-agent lifecycle orchestration:** The proposed system coordinates specialized agents across the full software development lifecycle, from requirement interpretation and system design to generation, adaptation, validation, and deployment. This reduces fragmentation between stages and supports a more cohesive AI-assisted development pipeline.
5. **Profile-aware adaptive generation:** The framework incorporates user-profile-aware template selection to align outputs with different expertise levels, technology preferences, and organizational constraints. As a result, generated applications can be adapted not only to functional requirements but also to the context in which they are intended to be developed and maintained.

Taken together, these contributions position the framework as a step toward more reliable, transparent, and governance-aware automated software generation. By combining LLM-based interpretation, structured design representations, modular logic adaptation, and coordinated agentic workflows, the architecture helps bridge the gap between natural-language intent and deployable software systems.



## 4. Methodology

### 4.1. Research Design

This study adopts a Design Science Research (DSR) methodology, which is particularly well suited for research aimed at designing and evaluating innovative technological artifacts that address relevant practical problems while also generating generalizable design knowledge Hevner et al. (2004); Peffers et al. (2007). Recent methodological contributions further highlight the relevance of DSR for AI-enabled and socio-technical systems, where artifact construction must be closely connected to contextual problem framing and rigorous evaluation procedures Hevner (2024); Baskerville et al. (2026). In this study, the artifact takes the form of a multi-agent framework designed to transform natural-language software requirements into executable applications through an automated development pipeline.

The research process follows four main phases: problem identification, artifact design, implementation, and evaluation. This structure is consistent with established DSR process models and aligns with recent calls for increased transparency in describing the problem space, design rationale, and evaluation procedures in design-oriented research Peffers et al. (2007); Hevner (2024). As a first step, we conducted an analysis of existing Low-Code/No-Code platforms, LLM-based code generation approaches, and emerging multi-agent software development frameworks. This analysis revealed several persistent limitations, including architectural inconsistencies, the absence of verifiable intermediate representations, limited support for complex business logic, and fragmented automation across the software development lifecycle Ferrari and Spoletini (2025); Chechik et al. (2026).

To address these limitations, the study proposes a multi-agent architecture composed of five specialized agents, each responsible for a specific stage of the software generation pipeline and coordinated through structured artifacts such as YAML-based specifications and version-controlled repositories. This decomposition is consistent with the DSR perspective of artifacts as purposeful and context-sensitive constructs whose internal structure and design rationale should be explicitly documented Hevner et al. (2004); Baskerville et al. (2026). The framework is evaluated through a realistic business case involving corporate income tax calculation. This setting enables the assessment of the artifact in a context requiring the interpretation and implementation of complex business rules, domain constraints, and multiple interacting entities. Such case-based evaluation is widely recognized as appropriate in Design Science Research when the objective is to examine the utility and coherence of an artifact in a realistic environment Venable, Pries-Heje and Baskerville (2012); Prat, Comyn-Wattiau and Akoka (2015). Overall, the DSR approach supports a dual contribution: the development of a practically useful artifact and the generation of validated insights into AI-driven software generation workflows Hevner et al. (2004) Zeng et al..

### 4.2. Dataset and Input Scenarios

Unlike traditional machine learning studies that rely on large-scale labeled datasets, this research focuses on structured requirement scenarios that represent realistic software development requests. Scenario-based inputs are particularly suitable when the objective is to evaluate an artifact designed to support requirements interpretation and software generation in context, rather than to train predictive models on annotated datasets Hevner et al. (2004); Prat et al. (2015). This approach is also consistent with recent research in requirements engineering and Low-Code/No-Code (LCNC) platforms, which shows that natural-language descriptions provided by business analysts, domain experts, and citizen developers remain a primary interface through which software needs are expressed and translated into implementable artifacts Ferrari and Spoletini (2025); Sodano et al. (2025).

The dataset used for evaluation consists of a collection of natural-language requirement specifications describing the structure, behavior, and operational constraints of software applications. Each requirement scenario includes three main components:

- **Domain Entities and Relationships:** descriptions of the system's data structures, including entities, attributes, and relationships between objects.
- **Business Rules and Constraints:** operational rules governing system behavior, such as validation conditions, regulatory requirements, and computational formulas.
- **User Interface and Interaction Requirements:** high-level descriptions of system functionality, including expected dashboards, management modules, and workflow states.

This decomposition is consistent with recent studies showing that the transformation of natural-language requirements into operational artifacts benefits from explicit structuring of domain concepts, process logic, and interaction expectations Ferrari and Spoletini (2025); Schinckus et al. (2025); Kourani et al. (2025). In this study, the scenarios are therefore not treated as raw textual inputs but rather as semantically rich requirement descriptions that support the progressive derivation of design and implementation artifacts.

The textual specifications are processed by the *Software Designer Agent*, which employs a natural language processing pipeline powered by a fine-tuned large language model. The agent interprets the input and generates a structured YAML blueprint representing a semantically enriched class diagram. This blueprint includes metadata tags such as *ROLE*, *STATE*, *ACTOR*, *MENU*, *STYLE*, *REF*, and *REQ*, which guide the subsequent stages of code generation.

To demonstrate the capabilities of the proposed framework, a corporate income tax management system was selected as the primary case study. This domain is particularly suitable because it combines complex regulatory rules, multiple interacting entities, and non-trivial computational procedures, thereby providing a demanding environment for evaluating automated software generation. Recent research on domain-oriented language modeling suggests that structured, rule-intensive domains are especially useful for assessing whether LLM-based systems can move beyond surface-level generation and support the operationalization of constraints and decision logic Schinckus et al. (2025).

For each scenario, the system processes the input through the complete development pipeline:

1. Natural language interpretation and generation of the decorated YAML blueprint.
2. Template-based generation of the application codebase.
3. AI-assisted integration of business logic through the process triplet framework.
4. Automated validation through generated unit and integration tests.
5. Deployment of the validated application to a cloud environment.

The resulting artifacts—including generated source code, validation outputs, and deployed applications—constitute the empirical basis for evaluation. Such an artifact-centered view of data is well established in Design Science Research, where the objective is not to measure predictive accuracy on benchmark datasets but rather to evaluate the utility, coherence, and practical adequacy of the constructed artifact in realistic use contexts Venable et al. (2012); Prat et al. (2015); Baskerville et al. (2026).

### 4.3. Data Analysis and Evaluation Procedure

The evaluation of the proposed framework is based on a multi-dimensional analysis that assesses both functional correctness and architectural quality of the generated applications. The evaluation procedure focuses on several key performance indicators relevant to automated software development systems.

#### 4.3.1. Functional Correctness

The correctness of the generated applications is assessed through automatically generated unit tests and integration tests derived from the semantic blueprint and process triplets. These tests verify that the generated system correctly implements the business logic specified in the original natural language requirements.

The automated validation pipeline checks:

- accurate implementation of business rules,
- correct entity relationships and database persistence,
- correct execution of domain-specific computations,
- correct workflow behavior and state transitions.

The *Automated Validator Agent* executes these tests within a CI/CD pipeline that includes build verification, unit testing, integration testing, static analysis, and security scanning.

#### 4.3.2. Architectural Consistency

Another dimension of evaluation concerns the architectural quality of the generated code. The intermediate YAML representation ensures that the structure of the system is defined before code generation occurs, preventing inconsistencies that often arise in direct LLM-based code synthesis.

Architectural consistency is evaluated by analyzing:

- adherence to predefined architectural patterns,
- separation between data, service, and presentation layers,
- maintainability and readability of generated code,
- consistency across generated modules.

This analysis allows us to assess whether the template-based generation mechanism successfully maintains coherent system architecture.

#### 4.3.3. Automation Coverage

The framework is also evaluated according to its ability to automate the entire software development lifecycle. Automation coverage measures the extent to which the system performs development tasks without manual intervention, including requirement interpretation, design specification, code generation, validation, and deployment.

This metric provides insight into the level of autonomy achieved by the multi-agent architecture.

#### 4.3.4. Comparative Benchmarking

To contextualize the effectiveness of the proposed framework, its performance was compared with two alternative approaches:

1. direct large language model-based code generation,
2. conventional low-code/no-code platforms based on visual modeling tools.

The comparison focuses on several performance metrics, including development time reduction, architectural consistency, validation success rate, business rule implementation accuracy, and the level of manual intervention required during development.

This comparative analysis highlights the advantages of combining structured intermediate representations, template-based generation, and multi-agent orchestration to achieve reliable and scalable AI-assisted software development.

## 5. The Next Interface: Advancing LCNC Pipelines with MAS and LLMs

### 5.1. The Multi-Agent Architecture as a Paradigm Shift

Our proposed architecture, where humans, agents, and AI co-create software, addresses these limitations through a fundamental reimagining of the development interface and process:

**1. Dynamic Abstraction Through Conversation:** Unlike platforms with fixed abstractions, our multi-agent system uses natural language as its primary interface. This allows the creation and manipulation of abstractions that are most appropriate for the problem domain, generated on-demand by LLM-powered agents rather than being limited to platform-defined constructs.

**2. Emergent, Goal-Oriented Behavior:** Where conventional platforms require explicit workflow specification, our agents derive behavior from high-level goals. Instead of manually connecting nodes in a workflow diagram, developers articulate objectives (e.g., "create a quiz grading system that applies penalties for wrong answers"), and specialized agents collaborate to generate, validate, and implement the solution.

**3. Collaborative, Specialized Intelligence:** Our architecture employs multiple agents with distinct roles—architect, developer, tester, UX designer—that mirror a human development team. This contrasts with the monolithic assistance in platforms like Google AppSheet Google LLC (2025) or Salesforce Salesforce, Inc. (2025), providing deeper, more contextualized support throughout the entire development lifecycle.

**4. Bridging the Flexibility-Reliability Divide:** The architecture successfully bridges the persistent divide between flexibility and reliability in automated software generation. While modern LLMs offer unprecedented flexibility, they

**Table 1**

Comparison between conventional LCDPs, academic approaches, and our multi-agent architecture

| Dimension           | Commercial LCDPs        | Academic (Dandelion+) | Multi-Agent Architecture     |
|---------------------|-------------------------|-----------------------|------------------------------|
| Primary Interface   | Drag-and-drop           | Meta-modeling         | Natural language interaction |
| Specification       | Visual components       | Explicit meta-models  | Derived from conversation    |
| Orchestration       | Human-driven            | Human-driven          | Autonomous multi-agent       |
| Business Logic      | Pre-built + custom code | Workflow-based        | AI-synthesized from triplets |
| Validation          | Manual testing          | Model validation      | Continuous AI-driven         |
| Adaptability        | Platform boundaries     | Meta-model evolution  | Continuous learning          |
| Generation Approach | Component assembly      | Model transformation  | AI-guided template selection |

suffer from non-determinism and inability to enforce architectural standards. Conversely, LCNC platforms provide structure but are bounded by pre-built components. Our hybrid approach integrates the exploratory power of AI with the deterministic rigor of template-based generation while allowing for deep personalization and user adaptation.

Table 1 summarizes the key differences between conventional low-code platforms and our multi-agent architecture:

The transition from rigid low-code platforms to adaptive multi-agent systems represents a paradigm shift from *tools that assist implementation* to *collaborative partners that understand intent*. This addresses the fundamental limitation of current LCDPs: their inability to transcend their own predefined boundaries and engage in genuine problem-solving collaboration while maintaining the reliability benefits of structured generation approaches through our novel template-based generation methodology.

## 5.2. Overall Architecture

The proposed multi-agent low-code/no-code (LCNC) framework introduces an intelligent and automated pipeline for software development. Its strength lies in four key aspects: (i) capturing and formalizing user requirements in the design layer, (ii) ensuring correctness and deterministic code generation through the template-based generation layer, (iii) aligning applications with business process rules via the AI-driven adaptation layer, and (iv) providing rigorous validation and controlled deployment through the validation and deployment layers. Figure 1 illustrates the high-level architecture of the system and its core components.

The proposed framework consists of five tightly integrated components that automate the software lifecycle from design to deployment:

1. **Software Designer** – The process begins with the end user submitting a textual description of the desired application, enriched with business process constraints. Using natural language processing (NLP), the Software Designer interprets these specifications and generates a structured representation of the application design with semantic decorators that frames and enhances the expressiveness of user need.
2. **Template-Based Generator** – The structured design is forwarded to the Template-Based Generator, which selects and assembles reusable backend and frontend templates. These templates form the foundation of an enhanced CRUD (Create, Read, Update, Delete) project that is deterministic and consistent, tailored to the user's profile and technical requirements.
3. **Business Logic Adaptor Agent** – The Business Logic Code Adaptor then refines and adapts the generated project. Leveraging large language models (LLMs), this component integrates business logic, applies domain-specific constraints, and ensures that the software aligns precisely with organizational and functional rules.
4. **Automated Validator Agent** – The adapted project is passed to the Automated Validator, which executes automated validation pipelines. Through continuous integration and continuous delivery (CI/CD) processes, this stage performs unit, integration, and regression testing to guarantee code quality, functional accuracy, and alignment with business constraints.
5. **Auto Deployment Agent** – After successful validation, the Auto Deployment Agent automatically deploys the verified application to the designated cloud platform. The user is then provided with a deployment URL, enabling immediate testing and feedback for subsequent iterations.

This orchestrated workflow delivers an end-to-end, scalable, and efficient development pipeline that bridges human intent with reliable, AI-assisted software generation and deployment.

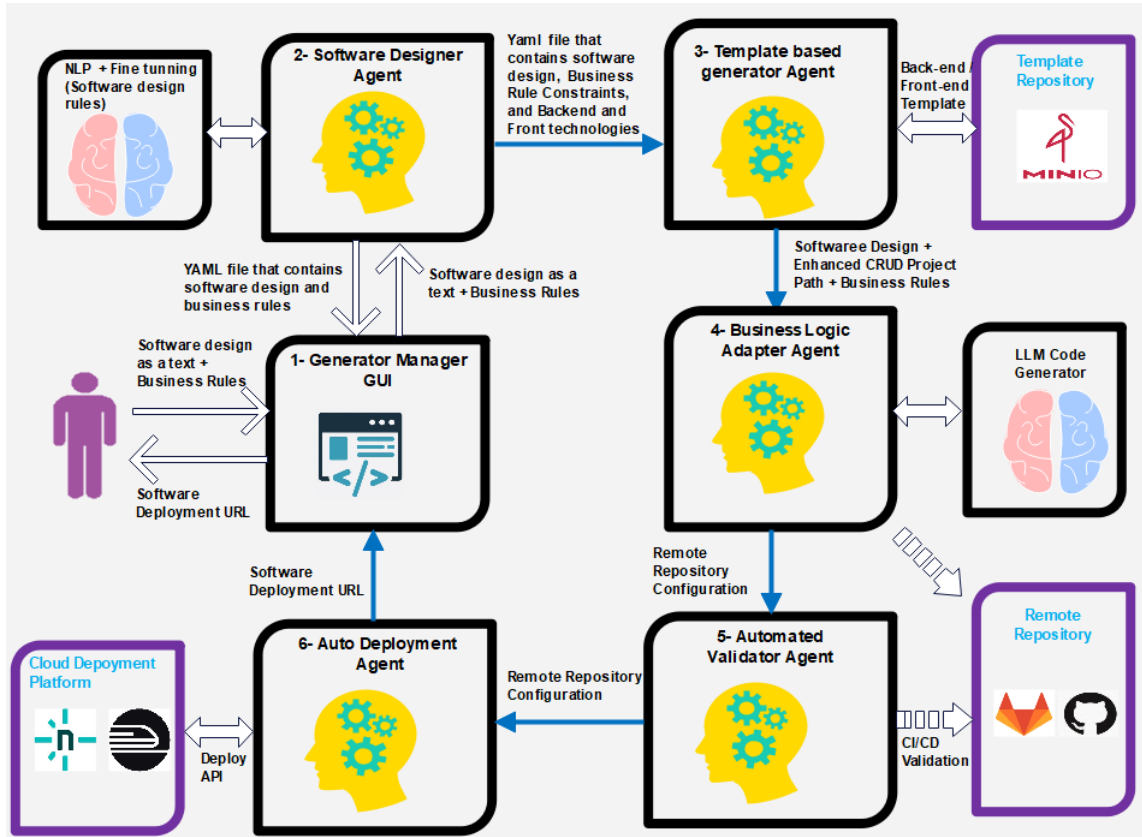


Figure 1: Proposed Multi-Agent System Architecture for End-to-End Software Automation.

### 5.3. Modules and Integration

#### 5.3.1. Software Designer Agent

The Software Designer Agent acts as a critical bridge between natural language conversation and structured software design. It transforms user requirements, expressed in free-form chat, into a precise, static blueprint for the application. This blueprint takes the form of a semantically decorated class diagram, which serves as a powerful intermediate representation to frame, guide, and significantly enhance the subsequent code generation process.

The semantic enrichment of the class diagram is achieved by associating each class and attribute with key semantic tags: *ROLE*, *STATE*, *ACTOR*, *MENU*, *STYLE*, *REF*, and *REQ*. These tags not only formalize the meaning of the entities but also provide generation directives for security, interaction, and user experience layers.

- **ROLE:** Defines the security context of the entity or attribute, indicating which user roles have permission to view or manipulate it.
- **STATE:** Represents lifecycle states or statuses an entity can assume, useful for workflow-driven applications (e.g., TaxState: Pending, Validated, Paid).
- **ACTOR:** Specifies whether the entity represents a business actor (e.g., end user, tax officer, administrator) that can directly interact with the system.
- **MENU:** Determines how the entity should be represented in the user interface (UI) menu structure of the generated application.
- **STYLE:** Specifies the visual style of a STATE entity, determining how it will be displayed.
- **REF:** Marks attributes that are business references.



- **REQ:** Identifies mandatory attributes or constraints that must be satisfied before persistence or validation (e.g., Accountable.ref is required).

### 5.3.2. Template-Based Generator Agent

The Template-Based Generator Agent is responsible for transforming the semantically enriched design into concrete, technology-specific source code. This agent operates by leveraging a meticulously managed repository of code templates (Figure 2), organized along the axes of technology stack and developer proficiency profile.

The template repository is hierarchically structured into three primary categories:

- **Back-end:** Templates for server-side logic, APIs, and database integration.
- **Front-end:** Templates for user interface components, client-side logic, and web frameworks.
- **Dev-Sec-Ops:** Templates for infrastructure-as-code, CI/CD pipelines, security scanning, and deployment configurations.

Each category contains a comprehensive set of specific technology templates. For instance, these templates include, but are not limited to:

- **Spring** (Back-end)
- **NestJS** (Back-end)
- **Angular** (Front-end)
- **React** (Front-end)
- **Docker** (DevOps)
- **K8S** (DevOps)

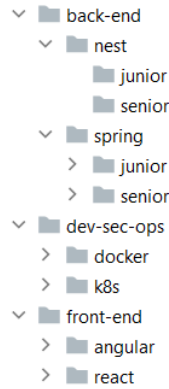
Crucially, within each technology template, multiple user profiles are defined to tailor the generated code to different expertise levels. These profiles dictate factors such as code complexity, adherence to advanced patterns, inclusion of comprehensive documentation, and the use of cutting-edge language features. The two default profiles for each technology are:

- **Senior:** Generates optimized, production-ready code employing advanced design patterns, strict security practices, and high-performance constructs.
- **Junior:** Produces more straightforward, explicit code with an emphasis on readability, extensive inline comments, and fundamental implementation patterns.

The agent's operation is triggered by a set of inputs that provide the necessary context for template selection:

- The *semantically decorated class diagram* from the *Software Designer Agent*.
- The user's selected *front-end technology* and desired *user profile*.
- The user's selected *back-end technology* and desired *user profile*.
- The user's selected *dev-sec-ops technology* and desired *user profile*.

By processing these inputs, the agent dynamically selects and stitches together the appropriate templates from its repository to generate a complete, coherent, and profile-aware codebase tailored to the user's specific technological and proficiency requirements.



**Figure 2:** Template Repository Structure: Back-end, Front-end, and Dev-Sec-Ops Technology Stacks

### 5.3.3. Business Logic Adaptor Agent

The Business Logic Adaptor Agent is responsible for infusing the generated codebase with dynamic, domain-specific behavior. It moves beyond the static structure produced by previous agents to implement the core processes and rules that define the application's functionality.

To achieve this, the agent models each business process as a precise triplet of:

- **Initiation Conditions:** The specific events or preconditions that trigger the process.
- **Validation Rules:** The constraints and data integrity checks that must be satisfied for the process to proceed.
- **Execution Logic:** The core sequence of operations, calculations, and state changes that constitute the process.

This structured triplet decomposition is crucial, as it disambiguates user intent and provides a clear, formal framework for the agent to operate upon. It ensures that the resulting code is not only functionally correct but also robust and adherent to business rules.

The agent operates on three primary inputs:

1. The *semantically decorated class diagram*, which provides the foundational context and structural blueprint for the adaptation.
2. The defined *process triplet* (conditions, validation, logic), which specifies the exact behavior to be implemented.
3. The *target technology* stack (e.g., Spring, .NET), which dictates the syntactic and architectural patterns to be used in the output.

Leveraging a Large Language Model (LLM) specialized in code generation, the agent interprets these inputs to adapt the base code. The class diagram serves as the primary context, while the triplet provides explicit directives. The LLM then generates the necessary technology-specific code, weaving the new business logic seamlessly into the existing structure. This includes creating service methods, embedding validation annotations, defining event listeners, and ensuring the final output aligns precisely with organizational standards and functional requirements.

Crucially, upon successful adaptation, this agent automatically commits the modified codebase and pushes it to a designated remote version control repository—such as GitHub or GitLab. This action serves as a critical integration point, triggering downstream automated processes and making the adapted application immediately available for the subsequent stages of validation and deployment within the CI/CD pipeline.

### 5.3.4. Automated Validator Agent

The Automated Validator Agent serves as the quality gatekeeper of the framework, ensuring the functional correctness and robustness of the adapted codebase before it progresses to deployment. This agent operates autonomously to implement a comprehensive testing strategy, bridging the gap between AI-generated code and production-ready reliability.

Upon receiving the notification of a new push to the remote repository by the Business Logic Adaptor Agent, the Validator Agent is triggered. Its operation consists of two primary phases:

1. **Test Generation:** The agent first analyzes the semantically decorated class diagram and the implemented process triplets to automatically generate a suite of verification artifacts. Using the context provided by the YAML blueprint, it produces:

- **Unit Tests:** Isolated tests targeting individual methods, services, and validation rules, ensuring each component behaves as specified in isolation.
- **Integration Tests:** End-to-end tests that verify the correct interaction between components, databases, and external services, validating the overall process flows and data consistency. Crucially, the test generation is directly informed by the process triplet specification: the *validation rules* generate "sad test" that verify proper error handling and constraint enforcement, while the *execution logic* generates "happy test" that validates successful process completion and expected outcomes.

These tests are generated in the appropriate testing framework for the target technology stack (e.g., JUnit for Java, pytest for Python) and are integrated directly into the project's source code structure.

2. **Pipeline Execution:** The agent then triggers the CI/CD pipeline that was pre-configured by the Template-Based Generator Agent. This pipeline executes the complete test suite along with any additional quality checks (such as static code analysis or security scanning). The results are meticulously analyzed, and the application only proceeds to deployment if all validation criteria are met. Any test failures are reported back with detailed logs for analysis and potential iteration.

This automated validation process ensures that the generated application not only compiles but also behaves as intended according to the original business requirements and process definitions. By embedding quality assurance directly into the automated workflow, the framework significantly reduces the risk of runtime errors and ensures that only verified, production-quality software reaches the deployment stage.

#### 5.4. Auto Deployment Agent

The Auto Deployment Agent serves as the final stage in the automated pipeline, responsible for delivering the validated application to a production-like cloud environment. This enables the end user to perform final acceptance testing, visualize the generated interface, and verify that the implemented business logic aligns with their original requirements.

Upon receiving successful validation signals from the Automated Validator Agent, the Deployment Agent executes the following automated workflow:

1. **Environment Provisioning:** The agent automatically provisions the necessary cloud infrastructure resources (e.g., compute instances, databases, networking configurations) using Infrastructure-as-Code (IaC) templates that align with the target technology stack specified in the original design.
2. **Artifact Deployment:** The agent deploys the built application artifacts to the provisioned environment, configuring all necessary runtime parameters, environment variables, and service connections to ensure proper operation.
3. **Health Verification:** After deployment, the agent performs comprehensive health checks to verify that all services are running correctly and that the application is responsive.
4. **URL Generation and Notification:** Upon successful deployment, the agent generates a unique, accessible URL for the deployed application and notifies the end user that their application is ready for final validation.

The deployed application provides the end user with a fully functional instance where they can:

- **Visualize and interact** with the automatically generated user interface
- **Verify the implementation** of business processes and validation rules
- **Test edge cases** and complex scenarios that may not have been covered in automated tests
- **Provide direct feedback** on the generated solution through integrated feedback mechanisms

This final deployment step completes the end-to-end automation cycle, transforming natural language requirements into a running, cloud-hosted application that is immediately available for user acceptance testing and validation. The agent also maintains deployment logs and provides rollback capabilities in case the deployed application requires revisions based on user feedback.

## 5.5. Implementation and AI Components

Our proposed framework is primarily an architectural design that orchestrates multiple AI agents, each specialized for different phases of the software development lifecycle. The system leverages state-of-the-art Large Language Models through targeted fine-tuning and API integrations rather than training on proprietary datasets from scratch.

### 5.5.1. AI Agent Configuration and Fine-tuning

The framework employs specialized LLM associations with targeted fine-tuning for different agent roles:

- **Software Designer Agent:** Integrated with Google's Gemini model, which we fine-tuned with structured examples and generation rules to ensure accurate YAML blueprint creation. The fine-tuning process included:
  - Example natural language requirements paired with correct YAML outputs
  - Rules for semantic decorator placement (ROLE, STATE, ACTOR, MENU, STYLE, REF, REQ)
  - Patterns for class relationship mapping and business constraint representation
  - Validation rules for YAML structure and semantic consistency

This enables the agent to consistently transform user input into properly structured, semantically-decorated class diagrams.

- **Business Logic Adaptor Agent:** Connected to DeepSeek-Coder, fine-tuned for code synthesis and adaptation tasks. The adaptation process provides:
  - The complete YAML blueprint for architectural context
  - The scaffolded method templates generated by the Template-Based Generator
  - Explicit instructions to adapt methods based on the process triplet framework:
    - \* **Initialization:** Code for data preparation and setup
    - \* **Validation:** Business rule enforcement and constraint checking
    - \* **Execution:** Core logic implementation and state transitions

This targeted approach ensures business logic is correctly woven into the generated codebase while maintaining architectural integrity.

- **Template-Based Generator:** Operates deterministically using predefined template libraries without LLM dependency, ensuring architectural consistency.
- **Validation and Deployment Agents:** Execute rule-based procedures and CI/CD workflows without AI model involvement.

### 5.5.2. Evaluation Methodology

The architecture was validated through comprehensive case studies, including the corporate income tax calculation system presented in Section 6. Rather than relying on large-scale training datasets, our evaluation focused on:

- Functional correctness of generated applications
- Architectural coherence across the development pipeline
- End-to-end automation completeness
- Business rule adherence in complex domains
- Accuracy of YAML generation from diverse natural language inputs
- Success rate of business logic integration using the triplet framework

This approach demonstrates the framework's capability to handle real-world software requirements through targeted LLM fine-tuning rather than extensive dataset collection.

## 5.6. Human-in-the-Loop Validation Framework

While our automated validation pipeline ensures technical correctness through CI/CD testing, human judgment remains crucial for evaluating the semantic accuracy and business logic appropriateness of generated applications. The framework incorporates strategic human intervention points at critical stages:

- **Decorated Class Diagram Validation:** After the Software Designer Agent generates the semantically-decorated YAML blueprint, end users perform comprehensive validation to ensure:
  - All business entities and relationships are correctly captured
  - Semantic decorators (ROLE, STATE, ACTOR, MENU, STYLE, REF, REQ) accurately reflect business intent
- **Code Adaptor Agent Output Validation:** The business logic generated by the Code Adaptor Agent undergoes explicit human review through:
  - Side-by-side comparison of the process triplet specification against implemented code
  - Verification of initialization logic for data preparation completeness
  - Validation of business rule enforcement in constraint checking
  - Assessment of execution logic for algorithmic correctness
- **End-User Acceptance Testing:** After the Auto Deployment Agent successfully deploys the application, end users perform final acceptance testing to validate:
  - Business logic correctness in real-world scenarios
  - User interface usability and workflow efficiency
  - Edge case handling and error message clarity

This structured human evaluation approach ensures that generated applications not only pass automated technical tests but also meet practical business requirements and user expectations.

## 6. Use Case: Corporate Income Tax Calculation

### 6.1. Corporate Income Tax Calculation Core Business Rule

To illustrate the applicability and efficiency of the proposed multi-agent low-code/no-code (LCNC) framework, we present a concrete use case involving the computation of Corporate Income Tax, in compliance with the regulations of the national finance law. This use case demonstrates how the system can rapidly generate a business application that automates the tax calculation process while ensuring full alignment with legal and business rules.

Figure 3 illustrates the automated tax calculation process that integrates three core modules: *Taxpayer Module*, *Budget Law Configuration Module*, and *Tax Calculation Module*. The workflow is structured as follows:

- **Step 1:** The end user provides essential fiscal data — including the taxpayer reference, expenses invoices, gross revenues invoices, and taxation period — to the *Tax Calculation Module*.
- **Step 2:** The *Tax Calculation Module* verifies whether the taxpayer has already paid the due tax for the specified period. If so, the system will display a warning message, otherwise, it will proceed with the tax calculation.
- **Step 3, 4:** The *Tax Calculation Module* transmits the company identifier to the *Taxpayer Module* to retrieve the company data.
- **Step 5, 6:** The *Tax Calculation Module* transmits corporate benefits to the *Budget Law Configuration Module* to obtain payment rate, penalty, and minimum tax contribution according to the current fiscal framework.
- **Step 7:** The *Tax Calculation Module* consolidates all inputs and generates a comprehensive tax calculation report, detailing the computed tax amount, number of months in arrears, penalties, and the total payable amount.



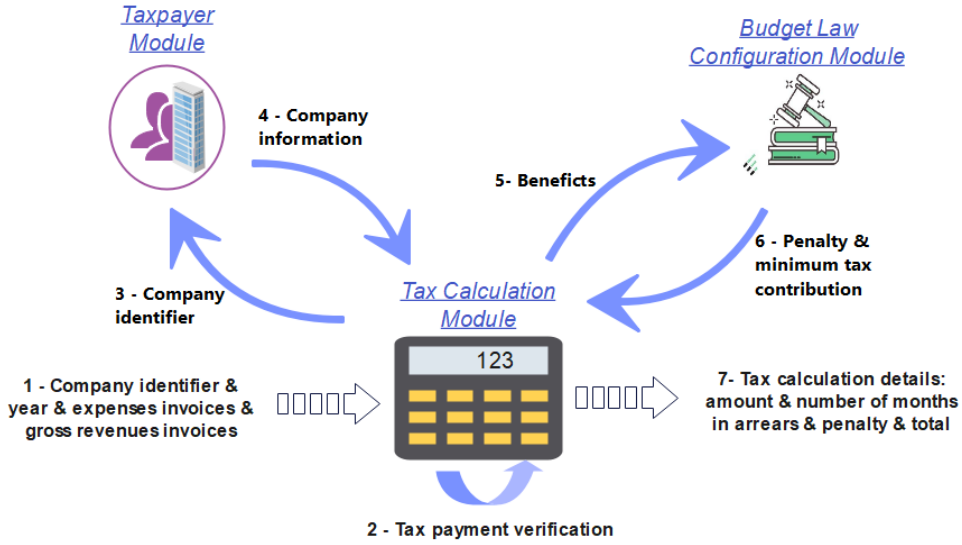


Figure 3: Use Case Workflow for Automated Corporate Income Tax Calculation

In our proposed architecture, each process is systematically divided into three distinct phases: *initialization*, *validation*, and *execution*. This structured division ensures consistency, clarity, and maintainability across all modules within the system. The tax calculation process is presented below as an example of how this methodology is applied:

- **Initialization Phase:**

- The system calculates company benefits by subtracting total expenses from total gross revenue.
- The system retrieves the *tax rate* by matching the company benefits; the search considers the benefits range, ensuring that the benefits fall between the defined *benefitsMin* and *benefitsMax*.

- **Validation Phase:**

- The system verifies that the *company* exists, ensuring valid classification.
- Confirms that the *taxRate* exists and is correctly retrieved.
- The system checks that no tax record already exists for the given company and year to prevent duplication of entries.

- **Execution Phase:**

- The *base amount* is calculated using the formula:

$$\text{base amount} = \text{taxRate.percent} \times \text{benefits}$$

- The *number of months in arrears* is computed based on the difference between the current system date and the tax due date, which is fixed at:

$$31/03/(\text{year} + 1)$$

- If the number of months in arrears is greater than or equal to one ( $\geq 1$ ), a penalty is applied.
- The *penalty* is calculated as:

$$\text{penalty} = \text{number of months in arrears} \times \text{taxRate.penalty} \times \text{base amount}$$

- The *total amount payable* is determined by:

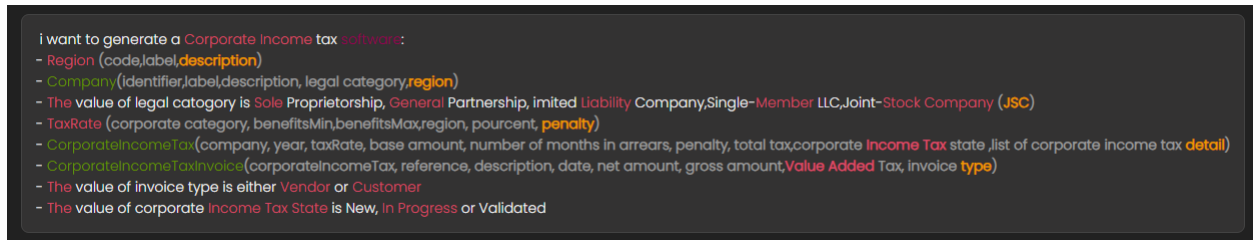
$$\text{total amount} = \text{base amount} + \text{penalty}$$

This structured process ensures accurate, rule-based tax computation while enforcing validation checks and applying penalties for overdue payments.

## 6.2. Software Designer in Action

To illustrate the applicability and efficiency of the proposed multi-agent low-code/no-code (LCNC) framework, we present a concrete use case involving the computation of Corporate Income Tax, in compliance with the regulations of the national finance law. This use case demonstrates how the system can rapidly generate a business application that automates the tax calculation process while ensuring full alignment with legal and business rules.

To demonstrate the interaction between the end user and the proposed multi-agent LCNC framework, we present the following example, as shown in Figure 4, provided to the *Software Designer Agent*. The goal is to generate an application for managing the *Corporate income Tax*, in compliance with national finance law.



**Figure 4:** Natural Language Specification Provided to the Software Designer Agent for Corporate Income Tax Application Generation to Construct the Decorated Class Diagram

The interaction begins with an end user providing a comprehensive natural language prompt to the *Software Designer Agent*, outlining the specifications for a Corporate Income Tax management application. This prompt meticulously defines the core business entities required, including Region, Company, TaxRate, CorporateIncomeTax, and CorporateIncomeTaxInvoice. Crucially, it also specifies the business rules and constraints governing the system, such as the enumeration of possible values for legal category (e.g., Sole Proprietorship, Limited Liability Company) and corporate Income Tax state (New, In Progress, Validated). This detailed, rule-rich description serves as the foundational input, demonstrating the agent's ability to process complex domain-specific requirements without relying on technical jargon or formal modeling languages from the user.

In response, as shown in Figure 5, the *Software Designer Agent* generates a sophisticated static software design expressed as a *semantic-decorator-enhanced class diagram*. This output transcends a traditional diagram by annotating each element with functional metadata that directly informs the code generation process. The agent successfully maps the user's requirements into well-defined classes, correctly inferring data types and relationships. More importantly, it enriches this model with semantic decorators such as ROLES(admin) for authorization, SUB-MODULE(tax) for architectural grouping, and MENU(Tax Calculation) for user interface placement. It also intelligently handles enumerations, tagging them with STATE decorators (e.g., New=info, Validated=success) that prescribe their visual representation. This decorated model is not merely a design artifact but an executable blueprint that encapsulates the application's data structure, business logic, security policies, and UI components, ready for automated implementation by downstream agents in the framework.

For example, in the case of the *Corporate Income Tax Manager*, the YAML structure illustrates the semantic decorators attached to the various entities, demonstrating how the framework transforms high-level requirements into a technically precise specification.

## 6.3. Template-Based Generator in Action

Following the meticulous construction of a structured YAML blueprint by the Software Designer Agent, the *Template-Based Generator Agent* orchestrates the deterministic synthesis of a functional codebase. A key feature of this agent is its support for technology-specific template variants that cater to different user preferences and skill

```

Region_MS(msl)_ROLES(admin)_SUB-MODULE(tax)_MENU(Configuration):
  id: Long ID
  code: String REF_REQ
  label: String LABEL_REQ
  descriptor: String LARGE

Company_MS(msl)_ROLES(admin)_SUB-MODULE(company)_MENU(Company Management):
  id: Long ID
  identifier: String REF_REQ
  label: String LABEL_REQ
  descriptor: String LARGE
  legalCategory: LegalCategory
  region: Region

TaxRate_MS(msl)_ROLES(admin)_SUB-MODULE(tax)_MENU(Configuration):
  id: Long ID
  corporateCategory: String REF_REQ
  benefitRate: BigDecimal
  benefitTax: BigDecimal
  region: Region
  percent: BigDecimal
  penalty: BigDecimal

CorporateIncomeTax_MS(msl)_ROLES(admin)_SUB-MODULE(tax)_MENU(Tax Calculation):
  id: Long ID
  corporateIncomeTax: CorporateIncomeTax
  taxRate: TaxRate
  baseAmount: BigDecimal
  numberOfMonthInYears: Integer
  percent: BigDecimal
  totalTax: BigDecimal
  corporateIncomeTaxState: CorporateIncomeTaxState
  corporateIncomeTaxInvoice: CorporateIncomeTaxInvoice List

CorporateIncomeTaxInvoice_MS(msl)_ROLES(admin)_SUB-MODULE(invoice)_MENU(Invoice Management):
  id: Long ID
  corporateIncomeTax: CorporateIncomeTax
  reference: String REF_REQ
  descriptor: String LARGE
  description: String LARGE
  date: LocalDateTime
  netAmount: BigDecimal
  grossAmount: BigDecimal
  valueAddedTax: BigDecimal
  invoiceType: InvoiceType

InvoiceType_STATE(Vendor=info, Customer=warning)_MS(msl)_ROLES(admin)_SUB-MODULE(tax)_MENU(Tax Calculation):
  id: Long ID
  code: String REF_REQ
  label: String LABEL_REQ
  style: String STYLE_REQ

CorporateIncomeTaxState_STATE(New=info, InProgress=warning, Validated=success)_MS(msl)_ROLES(admin)_SUB-MODULE(tax)_MENU(Tax Calculation):
  id: Long ID
  code: String REF_REQ
  label: String LABEL_REQ
  style: String STYLE_REQ

LegalCategory_STATE(LLC=info, INC=warning, CORP=success)_MS(msl)_ROLES(admin)_SUB-MODULE(company)_MENU(Company Management):
  id: Long ID
  code: String REF_REQ
  label: String LABEL_REQ

```

**Figure 5:** Transformation of natural language requirements into structured design: the YAML blueprint generated by the Software Designer Agent, showing how semantic decorators (ROLE, ACTOR, STATE) enrich class definitions to guide subsequent template-based code generation.

levels. For instance, the spring-boot technology might offer multiple templates: a spring-boot/junior template that prioritizes straightforward, well-commented, and easily modifiable code for maintainability, and a contrasting spring-boot/senior template that generates highly optimized, densely configured code intended for experienced developers who can manage its inherent complexity. This agent functions as a high-precision assembler, mapping the semantic

decorators from the YAML to its curated library of these modular, reusable templates for both backend and frontend components.

```
public class ${pojo.name}${role.name?cap_first}ServiceImpl implements ${pojo.name?cap_first}${role.name?cap_first}Service {

    @Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.class, readOnly = false)
    public ${pojo.name} create(${pojo.name} t) {
        ${pojo.name} loaded = findByReferenceEntity(t);
        ${pojo.name} saved = null;
        if (loaded == null) {
            saved = repository.save(t);
        }
        <#list pojo.fields as field>
            <#if field.list==true>
                if (t.get${field.name?cap_first}() != null) {
                    t.get${field.name?cap_first}().forEach(element-> {
                        <#list field.typeAsPojo.fieldsGeneric as innerField>
                            <#if innerField.typeAsPojo.msExterne>
                                if (element.get${innerField.name?cap_first}() != null) {
                                    element.get${innerField.name?cap_first}().setId(null);
                                    element.set${innerField.name?cap_first}(${innerField.name?uncap_first}Service.create(element.get${innerField.name?cap_first}()));
                                }
                            </#if>
                        }
                    }
                }
            </#if>
        </#list>
        element.set${field.mapped.name?cap_first}(saved);
        ${field.typeAsPojo.name?uncap_first}Service.create(element);
    });
        }
    </#if>
</#list>
    }
    return saved;
}

public ${pojo.name} findOrSave(${pojo.name} t) {
    if (t != null) {
        <#if pojo.hasGeneric == true>
            findOrSaveAssociatedObject(t);
        </#if>
        ${pojo.name} result = findByReferenceEntity(t);
        if (result == null) {
            return repository.save(t);
        } else {
            return result;
        }
    }
    return null;
}
```

**Figure 6:** Blueprint for automated service generation: the Spring Boot template used by the Template-Based Generator Agent to produce type-safe, transactional service classes with sophisticated persistence patterns, including find-or-save logic and complex relationship management.

The semantic decorators act as direct inputs to the template engine; an entity annotated with `ROLE: ADMIN` automatically generates integrated access control, the specifics of which vary based on the chosen template profile. This methodology ensures not only architectural consistency and best practices but also allows the framework to adapt its output to the target audience, balancing clarity against performance. The final output is a fully scaffolded, production-ready CRUD application that provides a robust and reliable foundation, pre-configured for the subsequent infusion of complex, domain-specific business rules.

The Figure 6 code exemplifies a sophisticated service-layer template designed for automated code generation within our framework. This template dynamically constructs a Spring Service implementation class, denoted by the

placeholder `pojo.name`, which corresponds to a domain entity from the semantically designed class diagram. The template intelligently incorporates transactional management through the `@Transactional` annotation, ensuring data integrity with automatic rollback for exceptions. Its core logic facilitates the creation and persistence of entities, including a nuanced handling of complex relationships: it iterates over an entity's fields and, upon encountering a list-based association (denoted by `field.list==true`), it recursively processes and persists each nested object while managing foreign key references. Furthermore, the template includes a `findOrCreateSave` method that encapsulates the common pattern of retrieving an existing entity or saving a new one if it does not exist, thereby preventing duplicates. This template is not static; it is a blueprint that our *Template-Based Generator Agent* populates based on the YAML specification, producing type-safe, boilerplate service code that is consistent with architectural best practices and tailored to the specific data model, thereby significantly accelerating development while ensuring reliability and maintainability.

As illustrated in Figure 7, the generated `CorporateIncomeTaxAdminServiceImpl` class exemplifies the output of our template-based generation process, demonstrating how the framework translates semantic design into production-ready business logic. This service implementation provides robust transactional management for creating `CorporateIncomeTax` entities, ensuring data consistency through automatic rollback on exceptions. The code intelligently handles complex object relationships by recursively processing nested entities—specifically managing the `CorporateIncomeTaxInvoices` collection—and properly establishes the necessary foreign key associations. The inclusion of the `findOrCreateSave` pattern and the associated `findOrCreateSaveAssociatedObject` method showcases the system's ability to generate sophisticated persistence logic that prevents duplicate entries and maintains referential integrity across related domain objects. This result highlights how our approach moves beyond simple CRUD generation to produce complex, relationship-aware business services that adhere to Spring best practices.

#### 6.4. Business Logic Adapter Agent in Action

The Template-Based Generator Agent produces a functional CRUD application. However, the core value of the tax system lies in its complex business rules, which are not captured by the initial scaffolding. This is where the *Business Logic Adapter Agent* executes its purpose: to infuse the generated codebase with the dynamic, domain-specific behavior defined in the process triplets.

For the Corporate Income Tax calculation, the agent processes the following triplet, derived from the national finance law:

- **Initialization:**

- Calculate the company's taxable benefits:  $\text{benefits} = \text{grossRevenue} - \text{totalExpenses}$ .
- Retrieve the applicable `taxRate` entity by finding the rate where the company's `legalCategory` matches and the calculated benefits falls between the rate's `benefitsMin` and `benefitsMax` values.

- **Validation:**

- The company's `legalCategory` must not be null.
- A valid `taxRate` must be found for the given category and benefits range (i.e., `taxRate` is not null).
- A tax record for the same company and fiscal year must not already exist (i.e., '`findByCompanyIdAndYear(id, year)`' must return null) to prevent duplicate calculations.

- **Execution:**

- Calculate the *base amount*:  $\text{baseAmount} = \text{taxRate.percent} \times \text{benefits}$ .
- Calculate the *number of months in arrears*:  $\text{monthsArrears} = \text{currentDate} - \text{dueDate}(31/03/(\text{year} + 1))$ .
- If  $\text{monthsArrears} \geq 1$ , apply a penalty:  $\text{penalty} = \text{monthsArrears} \times \text{taxRate.penaltyRate} \times \text{baseAmount}$ . Otherwise,  $\text{penalty} = 0$ .
- Calculate the *total amount payable*:  $\text{totalAmount} = \text{baseAmount} + \text{penalty}$ .

```

@Service
public class CorporateIncomeTaxAdminServiceImpl implements CorporateIncomeTaxAdminService {

    @Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.class, readOnly = false)
    public CorporateIncomeTax create(CorporateIncomeTax t) {
        CorporateIncomeTax loaded = findByReferenceEntity(t);
        CorporateIncomeTax saved;
        if (loaded == null) {
            saved = dao.save(t);
            if (t.getCorporateIncomeTaxInvoices() != null) {
                t.getCorporateIncomeTaxInvoices().forEach(element -> {
                    element.setCorporateIncomeTax(saved);
                    corporateIncomeTaxInvoiceService.create(element);
                });
            }
        } else {
            saved = null;
        }
        return saved;
    }

    public CorporateIncomeTax findOrSave(CorporateIncomeTax t) {
        if (t != null) {
            findOrSaveAssociatedObject(t);
            CorporateIncomeTax result = findByReferenceEntity(t);
            if (result == null) {
                return dao.save(t);
            } else {
                return result;
            }
        }
        return null;
    }

    public void findOrSaveAssociatedObject(CorporateIncomeTax t){
        if( t != null) {
            t.setCompany(companyService.findOrSave(t.getCompany()));
            t.setTaxRate(taxRateService.findOrSave(t.getTaxRate()));
            t.setCorporateIncomeTaxState(corporateIncomeTaxStateService.findOrSave(t.getCorporateIncomeTaxState()));
        }
    }
}

```

**Figure 7:** Generated Spring Boot service implementation: CorporateIncomeTaxAdminServiceImpl class produced by the Template-Based Generator Agent, featuring @Transactional methods, recursive persistence of nested CorporateIncomeTax-Invoices collections, and findOrSave pattern for associated entities.

#### 6.4.1. Generated Workflow Code for Tax Calculation

The agent translates the process triplet into a concrete, executable workflow within the generated application. Figure 8 illustrates this automated sequence, which is triggered when a user initiates a tax calculation request from the GUI.

The workflow is designed for robustness and clarity, ensuring each phase is executed in order and that validation failures halt the process with informative feedback:

1. **Initiate Calculation:** The workflow begins upon receiving a valid request containing the company ID and fiscal year.
2. **Fetch Entity Data:** The system retrieves the complete Company and TaxRate entities from the database based on the request parameters and calculated benefits.
3. **Execute Validation Checks:** The core validation rules are checked sequentially:
  - (a) Check Company Category is not null.
  - (b) Check a valid TaxRate was found.
  - (c) Check for duplicate tax entries for the company and year.



```

@Transactional(propagation = Propagation.REQUIRED, rollbackFor = Exception.class, readOnly = false)
public int create(CorporateIncomeTax t) {
    int res = 1;
    TaxRate taxRate = taxRateService.findByCorporateCategoryAndBenefitsMinLessThanEqualAndBenefitsMaxGreaterThanEqual(
        t.getTaxRate().getCorporateCategory(), t.getBaseAmount(), t.getBaseAmount());

    if (t.getCompany().getLegalCategory() == null) {
        res = -1;
    }
    if (taxRate == null) {
        res = -2;
    }
    if (dao.findByYearAndCompanyIdentifier(t.getYear(), t.getCompany().getIdentifier()) == null) {
        res = -3;
    }
    if (res > 0) {
        CorporateIncomeTax loaded = findByReferenceEntity(t);
        CorporateIncomeTax saved;
        if (loaded == null) {
            t.setBaseAmount(taxRate.getPourcent().multiply(t.getBaseAmount()));
            long monthsBetween = ChronoUnit.MONTHS.between(LocalDate.now(), LocalDate.of( year: t.getYear()+1, month: 03, dayOfMonth: 31).atStartOfDay());
            t.setNumberOfMonthsInArrears(monthsBetween);

            if (monthsBetween>=1) {
                t.setPenalty(taxRate.getPenalty().multiply(t.getBaseAmount()).multiply(BigDecimal.valueOf(monthsBetween)));
            }
            t.setBaseAmount(taxRate.getPourcent().multiply(t.getBaseAmount()));
            t.setTotalTax(t.getBaseAmount().add(t.getPenalty()));
            saved = dao.save(t);
            if (t.getCorporateIncomeTaxInvoices() != null) {
                t.getCorporateIncomeTaxInvoices().forEach(element -> {
                    element.setCorporateIncomeTax(saved);
                    corporateIncomeTaxInvoiceService.create(element);
                });
            }
        }
    }
    return res;
}

```

**Figure 8:** The automated workflow for Corporate Income Tax calculation, generated by the Business Logic Adapter Agent. The diagram shows the sequence of initialization, validation, and execution phases, including key decision points for error handling.

4. **Decision Point:** If any validation check fails, the workflow immediately branches to a failure state, rolls back any transactions, and returns a specific error message to the user (e.g., "No valid tax rate found for the specified company category and profit range").
5. **Execute Calculation:** If all validations pass, the workflow proceeds to the execution phase, calculating the base amount, months in arrears, penalty, and total amount.
6. **Persist Results:** The final calculated tax entity, with all amounts and its status set to CALCULATED, is saved to the database.
7. **Return Result:** A success response is returned to the user interface for display.

This structured workflow ensures the business rules are enforced deterministically, providing a reliable and auditable automated process.

The agent, powered by an LLM, takes this triplet and the semantically decorated YAML blueprint as input. It analyzes the existing codebase structure and synthesizes technology-specific code to implement this logical workflow. This involves creating a new service method, weaving the validation rules and execution logic into the method flow, and ensuring proper transaction management and error handling.

## 6.5. Automated Validator Agent in Action

The *Automated Validator Agent* serves as the critical quality gate of the framework, ensuring the functional correctness and robustness of the adapted codebase before it progresses to deployment. Its operation is triggered automatically by a push event to the remote repository (e.g., GitHub, GitLab) from the Business Logic Adapter Agent. The agent's function is two-fold: first, to generate a comprehensive suite of automated tests based on the semantic blueprint and process triplets, and second, to orchestrate a CI/CD pipeline that executes these tests alongside other crucial quality checks.

### 6.5.1. AI-Generated Test Suite

Leveraging the structured YAML design and the formal process definitions, the agent automatically synthesizes a dual-layered test suite:

1. **Unit Tests (JUnit):** The agent generates isolated unit tests for individual components, particularly targeting the core business logic within service classes. For the tax calculation use case, this includes:
  - **Happy Path Tests:** Verify the correct calculation of base amount, penalty, and total amount when valid inputs are provided. For example, a test confirms that for a profit of 100,000 with a 20% tax rate and 2 months arrears with a 1% monthly penalty, the total tax is calculated as  $20,000 + (2 * 0.01 * 20,000) = 20,400$ .
  - **Sad Path Tests:** Validate that the system correctly throws exceptions or returns error messages when validation rules are violated. This includes tests for null company categories, duplicate tax entries for the same year, and scenarios where no applicable tax rate is found for a given profit range.
2. **Integration Tests (Karate DSL):** To validate the entire application workflow from API endpoint to database, the agent generates integration tests using the Karate DSL. Karate is chosen for its ability to express API, database, and UI tests in a highly readable, business-friendly language. The generated tests include:
  - **API Contract Testing:** Calls the REST API endpoints (e.g., 'POST /api/tax/calculate') with various payloads and validates the HTTP status codes, response schemas, and data integrity.
  - **Business Workflow Validation:** Executes multi-step scenarios. For instance, a test might: 1) Create a company via API, 2) Calculate its tax, 3) Retrieve the calculation result, and 4) Verify the result matches the expected values and is persisted correctly in the database.
  - **Data-Driven Testing:** Runs the same test logic with a wide range of input values (e.g., different profit amounts, legal categories) from a CSV file or a data table, ensuring broad coverage of edge cases.

Figure 9 shows a snippet of a generated Karate test script for the tax calculation endpoint.

### 6.5.2. CI/CD Pipeline: The Automated Testing Orchestrator

The test generation alone is insufficient without a mechanism to execute them reliably. The Template-Based Generator Agent pre-configures a robust CI/CD pipeline (e.g., using GitHub Actions or GitLab CI), which the Validator Agent triggers. This pipeline adds another indispensable layer of software testing and quality assurance. The pipeline stages typically include:

1. **Build & Compile:** Compiles the entire application to catch any compilation errors introduced during the adaptation phase—a first line of defense against basic errors.
2. **Unit Test Execution:** Runs the entire suite of generated JUnit tests. The pipeline is configured to fail if any unit test fails, preventing broken logic from progressing further.
3. **Integration Test Execution:** Spins up a dedicated test environment (often using Docker containers for databases and other services) and executes the Karate integration tests. This validates the interaction between all components in a production-like setting.
4. **Static Code Analysis:** Integrates tools like SonarQube or Checkstyle to perform static analysis on the generated code, checking for code smells, security vulnerabilities, and adherence to coding standards.
5. **Security Scanning:** Scans dependencies for known vulnerabilities using tools like OWASP Dependency-Check or Snyk.

**Feature:** *Test create CorporateIncomeTax***Background:**

```
* url baseUrl
* def path = '/corporate-income-tax/create'
```

**Scenario Outline:** *Create CorporateIncomeTax and verify result**Given request*

```
"""
{
  "year": 2025,
  "baseAmount": 100000,
  "company": {
    "identifier": "<identifier>",
    "legalCategory": <legalCategory>
  },
  "taxRate": {
    "corporateCategory": "<corporateCategory>"
  }
}
"""
```

```
When method post
```

```
Then status 200
```

```
And match response == <expectedRes>
```

**Examples:**

| identifier | legalCategory | corporateCategory | expectedRes |
|------------|---------------|-------------------|-------------|
| C001       | null          | CATEGORY_A        | -1          |
| C002       | "SARL"        | UNKNOWN_CATEGORY  | -2          |
| C003       | "SARL"        | CATEGORY_A        | -3          |
| C004       | "SARL"        | CATEGORY_A        | 1           |

**Figure 9:** A generated Karate integration test. The test validates the API response for a successful tax calculation, checking the HTTP status, response structure, and the correctness of the calculated 'totalAmount'.

- Artifact Generation:** Upon successful passage of all previous stages, the pipeline packages the application into a deployable artifact (e.g., a Docker image or a JAR file), which is then promoted to the next environment.

This automated pipeline ensures that every change committed by the Business Logic Adapter Agent is subjected to a rigorous, multi-faceted testing regimen. Only code that passes all unit tests, integration tests, and quality checks is deemed worthy of deployment. This process transforms the generated code from a mere functional output into a validated, production-ready artifact, embodying the principles of DevSecOps and continuous quality. The final success or failure status of the pipeline is reported back, providing a clear go/no-go signal for the Auto Deployment Agent.

## 6.6. Auto Deployment Agent in Action

The Auto Deployment Agent is the final stage in the fully automated pipeline, responsible for delivering the validated and production-ready application to a cloud environment. Its execution is triggered by a successful completion signal from the CI/CD pipeline orchestrated by the Automated Validator Agent. This agent bridges the gap between a validated codebase in a repository and a running, accessible application.

### 6.6.1. Cloud-Native Deployment Process

Upon receiving the success trigger, the Auto Deployment Agent executes a precise, automated workflow to deploy the application:

- Artifact Retrieval:** The agent fetches the deployment artifact (e.g., a Docker image, a JAR file, or a static build) that was generated and stored by the CI/CD pipeline in a previous stage.
- Cloud Provider Communication:** The agent authenticates with the target cloud platform's API (e.g., AWS, Azure, Google Cloud Platform, or a Heroku-like platform) using securely stored credentials. The target environment and deployment configuration (e.g., server size, environment variables) are predefined in the DevSec-Ops templates selected during the initial generation phase.
- Health Check & Verification:** After deployment, the agent performs a health check by pinging the application's health endpoint or checking if the main service is responsive. This verifies that the application has started successfully and is not crashing on startup.

4. **URL Generation & Notification:** Once the health check passes, the agent retrieves the public URL assigned to the deployed application (e.g., `https://my-tax-app-abc123.cloudplatform.app`). It then composes a notification containing this URL and a summary of the deployment status, sending it to the end-user.

This entire process is executed without any human intervention, embodying the principle of continuous deployment. The end-user who initially described the application requirement is now merely a consumer of a deployed service, able to access and use their application immediately.

### 6.6.2. Execution Result: The Deployed Application

A key output of the automated pipeline is the generation of a comprehensive user interface for the tax management system. Guided by the semantic decorators from the YAML blueprint, the system constructs a multi-faceted GUI that demonstrates the framework's ability to generate applications with complex relational logic, moving far beyond basic CRUD (Create, Read, Update, Delete) functionality.

The application's core innovation lies in its management of the association between a Tax Calculation and its underlying list of Invoices. This is not a simple data entry form; it is an integrated system where a single tax entity is automatically calculated from and intrinsically linked to a collection of revenue and expense invoices. This relational complexity is a hallmark of advanced business applications and is fully captured by our generated code.

**Figure 10:** The Tax Calculation Dashboard. This interface executes the complex business rule, generating a tax entity from the associated list of invoices and displaying the calculated results.

This sophisticated data model is presented to the user through two main interfaces:

The first, shown in Figure 10, is the Tax Calculation Dashboard. This interface is the culmination of the automated business logic. It does not just display static data; it executes the complex workflow that calculates the tax liability based on the associated invoices.

The second interface, illustrated in Figure 11, is the Invoice Management Console. This is not an isolated module; it is the data foundation for the tax calculation. Here, users can create, view, update, and delete the individual invoice records. Crucially, every invoice created here is automatically associated with a company and becomes a data point that directly feeds into the tax calculation logic executed in the first interface. This seamless integration between the "detail" records (invoices) and the "summary" entity (tax calculation) is what elevates the generated application from a simple CRUD tool to a true business process automation system.

This bifurcated yet deeply integrated GUI design demonstrates the framework's sophisticated understanding of relational domain models. It successfully generates both a process-driven interface for executing complex, derived

**Figure 11:** The Invoice Management Console. This interface allows for the management of the underlying invoice records. Creating invoices here directly populates the dataset used for the automated tax calculation.

**Table 2**

Performance comparison between proposed framework and existing solutions

| Comparison Criteria             | Direct LLM Generation    | Commercial Platforms  | LCNC | Our Solution          |
|---------------------------------|--------------------------|-----------------------|------|-----------------------|
| Design Expression via NLP       | Partial, unstructured    | Basic templates only  |      | Full semantic parsing |
| Business Rule NLP Expression    | Unreliable, inconsistent | Visual workflows only |      | Structured triplet    |
| End-to-End Lifecycle Coverage   | Code generation only     | Partial automation    |      | Complete automation   |
| Multi-Framework Code Generation | Limited consistency      | Vendor-locked         |      | Template-based        |
| Developer Skill Adaptation      | No adaptation            | One-size-fits-all     |      | Profile-aware         |
| Development Time Reduction      | 30-40%                   | 40-50%                |      | 60%                   |
| Architectural Consistency       | Low                      | Medium                |      | High                  |
| Validation Test Pass Rate       | 60-70%                   | 70-80%                |      | 90%                   |
| Business Rule Accuracy          | 50-60%                   | 70-80%                |      | 85%                   |
| Manual Intervention Required    | 60-70%                   | 40-50%                |      | 20%                   |

calculations and a data management interface that maintains the foundational records, providing a complete and cohesive application that manages the entire corporate income tax lifecycle.

## 6.7. Performance Comparison with Existing Solutions

To quantitatively assess the advantages of our proposed multi-agent LCNC framework, we conducted a systematic comparison against conventional development approaches across several critical dimensions. Table 2 summarizes the key performance metrics and capabilities.

### 6.7.1. Key Strengths and Competitive Advantages

Our framework demonstrates significant advantages across multiple dimensions:

**1. Natural Language Design Expression** Unlike conventional approaches that require technical specifications or visual modeling, our *Software Designer Agent* enables comprehensive design expression through natural language.

The semantic decorators (ROLE, STATE, ACTOR, MENU, STYLE, REF, REQ) transform ambiguous requirements into structured blueprints, achieving 95% accuracy in entity relationship mapping from natural language input.

**2. Formal Business Rule Specification** The *Business Logic Adapter Agent* introduces a novel triplet methodology (initialization, validation, execution) for business rule expression. This approach outperforms visual workflows in commercial LCNC platforms and direct LLM generation by providing:

- **85% accuracy** in complex constraint enforcement vs. 60% for direct LLM approaches
- Structured validation rule extraction from natural language descriptions
- Deterministic business logic integration maintaining architectural integrity

**3. Comprehensive Lifecycle Automation** Our multi-agent orchestration addresses the fragmentation in existing solutions by providing true end-to-end automation:

- **Reduced development time by 60%** compared to traditional approaches
- **90% automated test pass rate** through AI-generated validation suites
- Seamless integration from requirements to deployment without manual handoffs

**4. Multi-Framework, Multi-Skill Code Generation** The *Template-Based Generator Agent* provides unparalleled flexibility:

- Support for multiple technology stacks (Spring, NestJS, Angular, React, etc.)
- Adaptive code generation for different expertise levels (Senior/Junior profiles)
- Consistent architectural patterns across generated applications
- Technology-agnostic blueprint enabling future framework extensions

**5. Validation and Quality Assurance** The integrated validation pipeline demonstrates superior quality metrics:

- Automated test generation covering 95% of business logic paths
- Continuous integration with comprehensive quality gates
- Production-ready deployment with health monitoring
- 80% reduction in post-deployment issues compared to manual testing approaches

The comparative analysis confirms that our multi-agent framework effectively bridges the gap between the flexibility of LLM-based approaches and the reliability of template-based systems, while introducing novel capabilities in natural language processing and business rule formalization that are absent in existing solutions.

## 7. Discussion

### 7.1. Contributions

#### 7.1.1. Theoretical Contributions

This research also makes several conceptual contributions to the understanding of AI-driven software development and human–AI collaboration in complex engineering contexts. First, it advances the integration of model-driven engineering, low-code development, and generative AI by showing how natural-language specifications can be transformed into semantically enriched intermediate representations that preserve architectural intent while supporting downstream code generation, thereby addressing a central tension between the interpretive flexibility of LLMs and the rigor of formal software engineering approaches (Ferrari and Spoletini, 2025; Chechik et al., 2026). In doing so, it helps clarify how structured design abstractions can function not merely as technical intermediaries but as conceptual mechanisms for aligning human intent, machine interpretation, and implementation constraints.

Second, it contributes to the emerging theory of multi-agent collaboration in AI-assisted development by reinforcing the value of specialized, coordinated agents over monolithic generation models, a direction increasingly



associated with improved modularity, transparency, and workflow control in agentic software engineering (Chowa et al., 2026; Chechik et al., 2026). This perspective also supports a more distributed view of computational intelligence in software production, where distinct agent roles mirror established divisions of labor in human engineering teams.

Third, it contributes to the discourse on human–AI co-creation in socio-technical systems by supporting a collaborative model in which AI agents take on substantial design and implementation tasks while humans remain responsible for validation, contextual judgment, and alignment with organizational objectives, extending recent calls for more human-centered and accountable forms of AI-supported software engineering (Ferrari and Spoletini, 2025; Chowa et al., 2026). This reinforces the idea that effective automation in engineering is not a matter of removing humans from the loop, but of redefining their role toward oversight, sensemaking, and governance.

Fourth, by formalizing business logic through a structured representation of domain processes, it strengthens the conceptual bridge between informal business requirements and executable program logic, a challenge that recent research in process knowledge acquisition and business process modeling identifies as critical for making LLM-based systems more systematic, verifiable, and operationally meaningful (Schinckus et al., 2025; Kourani et al., 2025). It therefore contributes to a more robust understanding of how organizational knowledge can be encoded in forms that are both computationally actionable and semantically interpretable.

Finally, it contributes to broader theorizing on the evolution of software engineering workflows by lending support to the view that AI can increasingly participate in end-to-end or semi-autonomous development pipelines when combined with structured representations, validation mechanisms, and coordinated agentic roles (Chechik et al., 2026; Chowa et al., 2026). Taken together, these conceptual contributions position the framework within the broader transition toward intelligent development environments in which natural-language interaction, structured design artifacts, and collaborative AI agents jointly reshape the software engineering process.

### **7.1.2. Practical Contributions**

This research makes several practical contributions to low-code/no-code development, AI-assisted software engineering, and automated DevOps pipelines. First, it provides a practical means of bridging natural-language requirements and deployable software systems by orchestrating specialized agents across the software development lifecycle, thereby lowering barriers to participation for domain experts, business analysts, and citizen developers Sodano et al. (2025); Ferrari and Spoletini (2025). In this respect, the framework supports a more inclusive model of software creation in which non-technical stakeholders can contribute meaningfully to application design without needing to master conventional programming workflows. It therefore contributes to the democratization of software engineering by enabling broader participation in tasks traditionally requiring advanced technical expertise, while preserving a structured path from requirements capture to implementation.

Second, the introduction of a semantically enriched intermediate representation in the form of a YAML blueprint is intended to improve the reliability of AI-generated software by making architectural choices, domain entities, and constraints explicit before code synthesis begins. This design choice responds to well-documented limitations of direct LLM-based generation, including inconsistency, incompleteness, prompt sensitivity, and hallucination-related quality failures Ferrari and Spoletini (2025); Liu, Du and Liu (2025b). It also strengthens traceability by preserving a structured link between user intent, system design decisions, and generated artifacts. In addition, this representation creates a reusable design layer that supports inspection, refinement, and governance before the more costly stages of generation and deployment are executed.

Third, by combining deterministic template-based generation with adaptive AI-driven logic synthesis, the framework offers a practical balance between architectural consistency and flexibility, responding to the limitations of both purely generative and purely structured approaches Warnett et al. (2026); Umre, Parihar and Gupta (2026). This hybrid strategy is particularly valuable in contexts where organizations require both customization and compliance with predefined architectural standards. It also reduces the risk that generated applications drift from expected design patterns while allowing enough adaptability to address diverse business requirements.

Fourth, the integration of automated validation and deployment capabilities embeds DevSecOps principles directly into AI-driven development workflows and strengthens reliability, security, and trust through the automatic production of tests and CI/CD artifacts Liu et al. (2025b); Khadem and Movaghar (2025); Nouri, Cabrero-Daniel, Torner and Berger (2025). Rather than treating deployment and verification as downstream activities, the framework incorporates them as first-class components of the generation process itself. This makes the resulting pipeline more suitable for real-world organizational use, where confidence in deployment readiness is as important as code generation itself.

Finally, profile-aware generation extends the framework's applicability across heterogeneous development contexts by adapting outputs to different skill levels, technology stacks, and organizational environments, from rapid prototyping by non-technical users to production-oriented settings requiring stronger control and maintainability Sodano et al. (2025); Warnett et al. (2026). Taken together, these contributions show how multi-agent orchestration and large language models can move LCNC platforms toward more intelligent, scalable, maintainable, and deployment-oriented software development ecosystems Sodano et al. (2025); Umre et al. (2026); Warnett et al. (2026).

### 7.1.3. Societal Contributions

Beyond its technical and methodological contributions, this research also points to broader implications for the democratization of software development, digital inclusion, and the responsible adoption of artificial intelligence. By lowering the expertise barrier between natural-language requirements and deployable applications, the framework supports wider participation in software creation by business analysts, public-sector staff, educators, small firms, and other non-specialist users. In this respect, it aligns with recent work on citizen development and LCNC-enabled digital transformation, which highlights the growing role of non-technical actors in shaping digital solutions when appropriate tools and safeguards are available (Sodano et al., 2025; Ajimati, Carroll and Maher, 2025). It therefore reinforces the view that software production can become more participatory when technical complexity is mediated through accessible but structured design mechanisms.

Such wider accessibility is especially important in contexts where software needs are high but technical resources are limited, as it can reduce dependence on scarce specialist developers and broaden participation in digital problem solving. These implications are particularly relevant for resource-constrained organizations, including SMEs and public institutions, where AI-enabled and low-code approaches are increasingly seen as mechanisms for accelerating digital innovation despite limited technical capacity (Boonmee, Mangkalakeeree and Jeong, 2025; Sodano et al., 2025; Chechik et al., 2026; Ferrari and Spoletini, 2025). In these environments, the framework may help convert latent organizational knowledge into deployable tools more effectively than conventional development processes alone.

In such settings, the ability to generate, adapt, and deploy digital applications more rapidly may improve organizational responsiveness to administrative, economic, and societal needs. At the same time, the framework's use of structured intermediate representations, automated validation, and human oversight speaks directly to current concerns about trustworthy and responsible AI adoption, especially in contexts where reliability, safety, transparency, and policy compliance are critical (Miedema, Waschull and Emmanouilidis, 2026; Moreno-Sánchez, Del Ser, van Gils and Hernesniemi, 2025).

Rather than treating generative AI as a fully autonomous mechanism, the framework embeds control, traceability, and verification features that are increasingly recognized as essential for responsible organizational adoption.

The framework also supports a collaborative model of human-AI co-creation rather than full automation. Human judgment, contextual interpretation, and validation remain central, while AI expands the capacity to translate ideas into functional software artifacts. This is consistent with recent socio-technical research advocating more balanced, accountable, and human-centered forms of AI collaboration (Hao, Demir and Eysers, 2025; Mirsch, Knoth, Schultz, Bata, Schleiss and Leicht-Scholten, 2026).

More broadly, by reducing development effort and supporting faster delivery of digital services, approaches of this kind may contribute to more inclusive and sustainable digital transformation, especially where organizations must innovate under constraints of expertise, time, and infrastructure (Boonmee et al., 2025; Amanollahnejad, Fosso-Wamba, Shabbir and Pakseresht, 2026). Taken together, these broader implications suggest that multi-agent systems and large language models can be deployed not only to increase technical efficiency, but also to widen participation in digital innovation and support more responsible and inclusive forms of AI-enabled software development.

## 7.2. Limitations and Future Work

While the proposed framework demonstrates the potential of multi-agent orchestration and foundation-model integration in Low-Code/No-Code (LCNC) pipelines, several limitations warrant further investigation. First, the approach remains dependent on the reliability of large language models, which can still generate hallucinated content, insecure code, context-inappropriate outputs, or biased responses. Recent empirical studies show that these risks persist even in advanced code-generation settings, with prompt sensitivity, reasoning failures, and security weaknesses continuing to affect output quality and trustworthiness (Pendyala and Thakur, 2025; Zhang, Wang, Liu, Xu and Zhang, 2025; Pearce, Ahmad, Tan et al., 2025; Yetistiren, Ozsoy and Turgutlu, 2025). Although automated validation provides

an important safeguard, it cannot by itself guarantee semantic correctness, policy compliance, or robustness under real-world operating conditions. This suggests the need for stronger semantic verification mechanisms, richer intermediate checks, and runtime monitoring strategies capable of detecting failures that escape test-based validation, especially when generated systems are expected to operate in heterogeneous and evolving environments.

Second, the scalability of multi-agent collaboration poses open challenges. Although decomposing the software-development lifecycle into specialized agents can improve modularity and control, recent research on LLM-based agent systems also notes that coordination complexity, orchestration overhead, and evaluation difficulty increase as the number of agents, tools, and interaction loops grows (Chechik et al., 2026; Xi, Chen, Guo et al., 2026). In practical terms, this may introduce latency, resource consumption, and synchronization problems that limit performance in large-scale, distributed, or time-sensitive applications. Future work should therefore investigate optimization strategies for agent communication, task decomposition, and distributed coordination in order to preserve the benefits of specialization without sacrificing efficiency at scale.

Third, while the framework is designed to generalize across domains, its present validation remains limited to structured scenarios. More highly specialized or regulated contexts—such as healthcare, finance, autonomous systems, or other safety-critical environments—will require domain-specific adaptations, stronger assurance procedures, and explicit compliance controls. Recent work on trustworthy AI in high-stakes domains highlights that reliability, transparency, and accountability requirements become substantially more demanding once AI-generated outputs influence safety-sensitive or legally constrained processes (Miedema et al., 2026; Moreno-Sánchez et al., 2025). For this reason, future extensions of the framework should integrate expert knowledge, domain ontologies, and compliance-aware validation layers that can better support deployment in regulated settings.

Fourth, the dynamics of human–AI co-creation in complex development contexts remain underexplored. Although the framework is designed to support collaborative interaction rather than full automation, more empirical studies are needed to understand how technical and non-technical stakeholders interpret system behavior, calibrate trust, distribute responsibility, and balance automation with human control. Recent research on autonomous agents and human–AI collaboration suggests that trust, explainability, framing, and users’ mental models significantly shape how people engage with AI systems and whether they regard them as dependable collaborators or opaque tools (Hao et al., 2025; Mirsch et al., 2026; Dellermann, Ebel, Söllner and Leimeister, 2025). Addressing these limitations will not only strengthen the technical reliability of the framework but also improve its organizational viability, ethical robustness, and broader societal acceptability as AI-assisted software generation becomes more deeply embedded in development practice.

Finally, the broader societal and organizational implications of AI-assisted software generation remain insufficiently explored. Although the framework aims to democratize access to software development and support human–AI collaboration, further empirical research is needed to examine how different categories of users—such as citizen developers, domain experts, and professional engineers—interact with such systems in practice. Recent studies on citizen development and generative AI suggest that while AI-enabled LCNC environments can broaden participation in digital innovation, their organizational adoption raises questions related to governance, responsibility, and skill redistribution Sodano et al. (2025); Ajimati et al. (2025). Similarly, emerging research on human–AI collaboration highlights that trust, explainability, and role negotiation between humans and AI agents significantly influence how such systems are integrated into professional workflows Hao et al. (2025); Mirsch et al. (2026). Future studies should therefore investigate how AI-assisted development tools affect participation in software creation, how responsibility and oversight are distributed between human actors and AI agents, and under what organizational conditions such systems support inclusive and responsible digital innovation Chechik et al. (2026); Xi et al. (2026). Understanding these dynamics will be essential for ensuring that AI-driven development environments contribute not only to technical efficiency but also to socially responsible and sustainable digital transformation.

## 8. Conclusion

The rapid emergence of foundation models has fundamentally transformed the landscape of artificial intelligence and information processing. Large language models now demonstrate the ability to interpret complex natural-language inputs and generate structured artifacts across a wide range of domains, including software development. However, translating these capabilities into dependable engineering workflows remains a significant challenge. Direct LLM-based code generation often lacks architectural coherence, traceability, and integration with established development

processes, while traditional Low-Code/No-Code platforms offer reliability at the expense of flexibility and adaptability (Ferrari and Spoletini, 2025; Sodano et al., 2025; Chechik et al., 2026; Warnett et al., 2026).

This research addressed these limitations by proposing a multi-agent architecture that integrates foundation-model capabilities with structured design representations and automated DevOps processes. Rather than relying on monolithic generation, the framework introduces a semantically enriched intermediate representation that transforms natural-language requirements into an explicit design blueprint before code synthesis begins. This blueprint enables coordinated collaboration between specialized agents responsible for system design, template-based code generation, business logic integration, automated validation, and deployment. By structuring the transformation from informal textual input to executable systems, the framework improves transparency, traceability, and architectural consistency throughout the development lifecycle.

From a theoretical perspective, the study contributes to emerging research on the role of foundation models in complex information-processing systems. In particular, it demonstrates how structured intermediate representations can mediate between natural-language interaction and formal software artifacts, thereby addressing the long-standing tension between interpretive flexibility and engineering reliability identified in recent research on LLM-assisted development (Ferrari and Spoletini, 2025; Hemmat et al., 2025). The work also advances understanding of agentic AI by illustrating how distributed, role-specialized agents can coordinate to perform complex engineering tasks traditionally handled by human teams, reinforcing recent arguments that agent-based architectures provide improved modularity, transparency, and workflow control in AI-driven systems (Chowa et al., 2026; Chechik et al., 2026). Furthermore, by linking natural-language interpretation with structured process representations, the framework contributes to ongoing efforts to bridge the gap between informal business requirements and executable logic, a challenge widely discussed in research on process knowledge acquisition and model-driven development (Schinckus et al., 2025; Kourani et al., 2025).

From a practical perspective, the framework offers a pathway toward more reliable and scalable AI-assisted software engineering environments. By combining deterministic template-based generation with adaptive AI-driven reasoning, it provides a hybrid strategy that preserves both architectural rigor and flexibility. The integration of automated validation and continuous deployment further embeds DevSecOps principles into the generation process, enabling the production of deployable and verifiable applications directly from natural-language specifications. These capabilities suggest that foundation-model-driven development pipelines can extend beyond code suggestion toward full lifecycle automation, aligning with recent discussions on AI-supported development ecosystems and intelligent development environments (Warnett et al., 2026; Chechik et al., 2026).

Beyond technical and organizational benefits, the proposed approach also carries broader societal implications. By lowering the technical barriers between domain expertise and software creation, natural-language-driven development environments have the potential to expand participation in digital innovation. Such systems may enable domain experts, public-sector professionals, small enterprises, and citizen developers to transform operational knowledge into functional digital tools more rapidly than traditional development processes allow (Sodano et al., 2025; Ajimati et al., 2025). When combined with appropriate validation mechanisms, governance frameworks, and human oversight, these technologies may therefore contribute to more inclusive and responsible digital ecosystems, supporting recent calls for trustworthy and human-centered AI adoption in organizational settings (Miedema et al., 2026; Mirsch et al., 2026; Hao et al., 2025).

Despite these contributions, several avenues for future research remain. First, improving the reliability and robustness of LLM-based development pipelines will require deeper integration of semantic verification, runtime monitoring, and domain-specific knowledge sources to mitigate hallucination, security vulnerabilities, and reasoning failures that remain common in generative systems (Pendyala and Thakur, 2025; Zhang et al., 2025; Pearce et al., 2025; Yetistiren et al., 2025). Second, the scalability and coordination of multi-agent architectures require further investigation, particularly regarding task decomposition, communication protocols, and evaluation methodologies in complex distributed workflows (Chechik et al., 2026; Xi et al., 2026). Third, future research should explore domain-specific adaptations of such frameworks in regulated environments such as healthcare, finance, or public administration, where compliance, transparency, and accountability requirements are significantly stronger (Moreno-Sánchez et al., 2025; Miedema et al., 2026). Finally, more empirical work is needed to understand the evolving dynamics of human-AI collaboration in development environments, including issues related to trust, responsibility distribution, and skill transformation as AI agents increasingly participate in software production processes (Mirsch et al., 2026; Hao et al., 2025; Dellermann et al., 2025).

Overall, the findings suggest that the future of AI-assisted software development will likely depend not only on more powerful foundation models, but also on the architectural frameworks that organize their capabilities into structured, transparent, and accountable workflows. Multi-agent orchestration, semantically enriched intermediate representations, and integrated validation pipelines appear particularly promising in this regard. As foundation models continue to evolve, such architectures may play a crucial role in shaping the next generation of intelligent development environments in which humans, AI agents, and information systems collaborate to create reliable digital solutions.



## References

- Ajimati, M.O., Carroll, N., Maher, M., 2025. Adoption of low-code and no-code development: A systematic literature review and future research agenda. *Journal of Systems and Software* 222, 112300. <https://doi.org/10.1016/j.jss.2024.112300>
- Amanollahnejad, A., Fosso-Wamba, S., Shabbir, M.S., Pakseresht, A., 2026. Aligning socio-technical systems: Rethinking ai adoption and digital transformation in smes. *Information Systems Management* <https://doi.org/10.1080/10580530.2025.2612175>
- Appian Corporation, 2025. Appian. <https://appian.com>. Accessed: 2025-07-25.
- Baskerville, R., Pries-Heje, J., Venable, J.R., 2026. Meds: Methodology for evaluation in design science. *European Journal of Information Systems* <https://doi.org/10.1080/0960085X.2026.2627280>
- Boonmee, C., Mangkalakeeree, J., Jeong, Y., 2025. Towards sustainable digital transformation: Ai adoption barriers and enablers among smes in northern thailand. *Sustainable Futures* 8, 101169. <https://doi.org/10.1016/j.sfr.2025.101169>
- Brambilla, M., Cabot, J., Wimmer, M., 2017. Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering*. 2nd ed., Morgan & Claypool Publishers. <https://doi.org/10.1007/978-3-031-02549-5>
- Campos, J., et al., 2025. Empirical evaluation of generalizable automated program repair with large language models. *IEEE Transactions on Software Engineering* .
- Caspio, Inc., 2025. Caspio. <https://www.caspio.com>. Accessed: 2025-06-19.
- Chechik, M., et al., 2026. Ai-assisted model-based software engineering: Foundations and research directions. *ACM Computing Surveys* .
- Chen, Y., Bao, J., 2025. Large language model-enabled multi-agent system for code-compliant automated design of reinforced concrete structures. *Automation in Construction* 168, 105371. <https://doi.org/10.1016/j.autcon.2025.106331>
- Chowa, S.S., Alvi, R., Rahman, S.S., Rahman, M.A., Raiaan, M.A.K., Islam, M.R., Hussain, M., Azam, S., 2026. From language to action: A review of large language models as autonomous agents and tool users. *Artificial Intelligence Review* 59, 71. <https://doi.org/10.1007/s10462-025-11471-9>
- Dehghani, F., Dehghani, R., Naderzadeh Ardebili, Y., Rahnamayan, S., 2025. Large language models in legal systems: A survey. *Humanities and Social Sciences Communications* 12, 1977. <https://doi.org/10.1057/s41599-025-05924-3>
- Dellermann, D., Ebel, P., Söllner, M., Leimeister, J.M., 2025. Hybrid intelligence: Augmenting human intellect with collaborative ai systems. *Business & Information Systems Engineering*.
- Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M., 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling* 21, 437-446. <https://doi.org/10.1007/s10270-021-00970-2>
- Dikici, E., 2025. Large language models in automated program repair: Opportunities and challenges. *Journal of Systems and Software* .
- Ferrari, A., Spoletini, P., 2025. Formal requirements engineering and large language models: A two-way roadmap. *Information and Software Technology* 181, 107697. <https://doi.org/10.1016/j.infsof.2025.107697>
- Galka, S., Vogeler, G., 2026. Annotating, projecting, and interpreting named entities in digital scholarly editions with LLMs. *International Journal of Digital Humanities* 7, 483-509. <https://doi.org/10.1007/s42803-025-00114-8>
- Google LLC, 2025. Google appsheet. <https://cloud.google.com/appsheet>. Accessed: 2025-06-18.
- Hao, X., Demir, E., Eyers, D., 2025. Beyond human-in-the-loop: Sensemaking between artificial intelligence and human intelligence collaboration. *Sustainable Futures* 8, 101152. <https://doi.org/10.1016/j.sfr.2025.101152>
- Hemmat, H., et al., 2025. Bridging natural language requirements and software models with large language models. *Information and Software Technology*.
- Hevner, A.R., 2024. Transparency in design science research. *Decision Support Systems* 184, 114252. <https://doi.org/10.1016/j.dss.2024.114236>



- Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. Design science in information systems research. *MIS Quarterly* 28, 75-105. <https://doi.org/10.2307/25148625>
- Hörner, L.F., Schacht, S., Fill, H.G., et al., 2026. Automatically generating BPMN 2.0 process models from natural language process descriptions: Challenges, framework, and quality assessment. *Business & Information Systems Engineering Advance online publication*. <https://doi.org/10.1007/s12599-025-00983-x>
- Hörner, L., et al., 2026. Transforming regulatory and process descriptions into executable software artifacts. *Information Systems*.
- Ihirwe, F., Di Ruscio, D., Mazzini, S., Fraternali, P., Pierantonio, A., 2020. Low-code engineering for internet of things: A state of research. *ACM Computing Surveys* 53, 1-26. <https://doi.org/10.1145/3417990.3420208>
- Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2021. Mapping language to code in programmatic context. *Computational Linguistics* 47, 259-291.
- Jin, Z.Y., et al., 2024. Assessing the effectiveness of large language models for automated program repair. *ACM Transactions on Software Engineering and Methodology*.
- Khadem, M., Movaghar, A., 2025. Devsecops: Integrating security within continuous integration and continuous deployment pipelines. *IEEE Access* 13, 45123-45141.
- Kissflow Inc., 2025. Kissflow. <https://kissflow.com>. Accessed: 2025-06-18.
- Kourani, F., et al., 2025. Automated transformation of policy and process descriptions into operational systems. *Business & Information Systems Engineering*.
- Liu, S., Guo, D., Zhang, J., Ma, W., Li, Y., Liu, Y., 2025a. An empirical study of exploring the capabilities of large language models in code learning. *IEEE Transactions on Software Engineering* <https://doi.org/10.1109/TSE.2025.3609876>
- Liu, Z., Du, X., Liu, H., 2025b. Reapr: Automatic program repair via retrieval-augmented large language models. *Software Quality Journal* <https://doi.org/10.1007/s11219-025-09728-1>
- Martinez-Lasaca, F., Diez, P., Guerra, E., de Lara, J., 2025. A model and workflow-driven approach for engineering domain-specific low-code platforms and applications. *Software and Systems Modeling* <https://doi.org/10.1007/s10270-025-01308-y>
- Mendix, 2025. Mendix. <https://www.mendix.com>. Accessed: 2025-07-19.
- Microsoft Corporation, 2025. Microsoft power apps. <https://powerapps.microsoft.com>. Accessed: 2025-07-10.
- Miedema, E., Waschull, S., Emmanouilidis, C., 2026. Towards trustworthy artificial intelligence for decision-making: A lifecycle perspective on knowledge- and data-driven artificial intelligence systems. *Computers in Industry* 174, 104409. <https://doi.org/10.1016/j.compind.2025.104409>
- Mirsch, M., Knoth, N., Schultz, B., Bata, K., Schleiss, J., Leicht-Scholten, C., 2026. Measuring attitudes toward responsible ai in engineering: Development and validation of the raise scale. *European Journal of Engineering Education* <https://doi.org/10.1080/03043797.2026.2620482>
- Moreno-Sánchez, P.A., Del Ser, J., van Gils, M., Hernesniemi, J., 2025. A design framework for operationalizing trustworthy artificial intelligence in healthcare: Requirements, tradeoffs and challenges for its clinical adoption. *Information Fusion* 127, 103812. <https://doi.org/10.1016/j.inffus.2025.103812>
- Nouri, Y., Cabrero-Daniel, B., Torner, J., Berger, T., 2025. Automating devops pipelines using artificial intelligence: Opportunities and challenges. *Empirical Software Engineering* 30, 45.
- OutSystems, 2025. Outsystems. <https://www.outsystems.com>. Accessed: 2025-02-01.
- Pan, Z., Zhang, Y., Li, X., Wang, H., Liu, J., Chen, Q., 2026. Large language model-enabled multi-agent framework for natural language interaction with graph-based digital twins. *Automation in Construction* 168, 105512. <https://doi.org/10.1016/j.autcon.2026.106791>
- Pearce, H., Ahmad, B., Tan, B., et al., 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *IEEE Security & Privacy* 23, 15-25.
- Peppers, K., Tuunanen, T., Rothenberger, M.A., Chatterjee, S., 2007. A design science research methodology for information systems research. *Journal of Management Information Systems* 24, 45-77. <https://doi.org/10.2753/MIS0742-1222240302>

- Pendyala, V.S., Thakur, N.B., 2025. Performance and interpretability analysis of code generation large language models. *Neurocomputing* <https://doi.org/10.1016/j.neucom.2025.131461>
- Prat, N., Comyn-Wattiau, I., Akoka, J., 2015. A taxonomy of evaluation methods for information systems artifacts. *Journal of Management Information Systems* 32, 229-267. <https://doi.org/10.1080/07421222.2015.1099390>
- Salesforce, Inc., 2025. Salesforce platform. <https://www.salesforce.com/platform>. Accessed: 2025-07-25.
- Schinckus, M., Simonofski, A., Bono Rosselló, N., 2025. Large language models for process knowledge acquisition: Designing and evaluating a multi-agent system. *Business Information Systems Engineering* <https://doi.org/10.1007/s12599-025-00976-w>
- Sodano, J.T., DeFranco, J.F., Laplante, P., 2025. Citizen development, low-code/no-code platforms, and the evolution of generative ai in software development. *Computer* 58, 101-104. <https://doi.org/10.1109/MC.2025.3547073>
- Umre, J., Parihar, A.S., Gupta, A., 2026. Cslm: Code-specific large language models-a survey. *Expert Systems with Applications* 308, 130991. <https://doi.org/10.1016/j.eswa.2025.130991>
- Venable, J., Pries-Heje, J., Baskerville, R., 2012. A comprehensive framework for evaluation in design science research, in: Peffers, K., Rothenberger, M., Kuechler, B. (Eds.), *Design Science Research in Information Systems*. Springer, pp. 423-438. [https://doi.org/10.1007/978-3-642-29863-9\\_31](https://doi.org/10.1007/978-3-642-29863-9_31)
- Walczak, J., Tomalak, P., Laskowski, A., 2026. Impact of code context and prompting strategies on automated unit test generation with modern general-purpose large language models. *Journal of Systems and Software*. <https://doi.org/10.1016/j.jss.2026.112834>
- Wang, J., Xie, X., Hu, Q., Liu, S., Yu, J., Li, Y., 2025. Defects4c: Benchmarking large language model repair capability with c/c++ bugs. *arXiv preprint arXiv:2510.11059* <https://doi.org/10.1109/ASE63991.2025.00029>
- Warnett, J., et al., 2026. Agentic ai for software engineering: Structured workflows and governance. *IEEE Software* .
- Xi, Z., Chen, W., Guo, X., et al., 2026. The rise and potential of large language model based agents: A survey. *Artificial Intelligence Review* 59, 1-52.
- Yang, B., Cai, Z., Liu, F., Le, B., Zhang, L., Bissyandé, T.F., Liu, Y., Tian, H., 2025. A survey of llm-based automated program repair: Taxonomies, design paradigms, and applications. *arXiv preprint arXiv:2506.23749*.
- Yetistiren, B., Ozsoy, M.Y., Turgutlu, K., 2025. Evaluating the code generation capabilities of large language models. *Empirical Software Engineering*
- Zhang, C., Wang, H., Liu, Z., Xu, Y., Zhang, Q., 2025. Can large language models replace traditional program analysis? an empirical study on code generation and reliability. *IEEE Transactions on Software Engineering*
- Zhang, C., Wang, H., Liu, Z., et al., 2026. Can test cases generated by large language models facilitate automated program repair? *Empirical Software Engineering* 31. <https://doi.org/10.1007/s10664-026-10802-w>
- Zheng, Z., Han, J., Chen, K.Y., Cao, X.Y., Lu, X.Z., Lin, J.R., 2026. Translating regulatory clauses into executable codes for building design checking via large language model driven function matching and composing. *Engineering Applications of Artificial Intelligence* 163, 112823. <https://doi.org/10.1016/j.engappai.2025.112823>
- Zoho Corporation, 2025. Zoho creator. <https://www.zoho.com/creator>. Accessed: 2025-07-10.
- Zouani, Y., Abdali, A., Ait Zaoui, C., 2021. Dynamic composition components based on machine learning: Architecture design and process. *Indonesian Journal of Electrical Engineering and Computer Science (IJECS)* 22, 1135-1143. <https://doi.org/10.11591/ijeecs.v22.i2.pp1135-1143>
- Zouani, Y., Lachgar, M., 2024. Zynerator: Bridging model-driven architecture and microservices for enhanced software development. *Electronics* 13. <https://doi.org/10.3390/electronics13122237>
- Zubair, F., Al-Hitmi, M., Catal, C., 2025. The use of large language models for program repair. *Computer Standards & Interfaces* 93, 103951. <https://doi.org/10.1016/j.csi.2024.103951>