Multi-agent architecture for legacy custom code migration*

Oleg Pozdnyakov[1,*,†], Anzhelika Parkhomenko[1,†]

[1] National University Zaporizhzhia Polytechnic, Zhukovskogo str., 64, Zaporizhzhia, 69011, Ukraine

Abstract

This paper addresses the problem of automating legacy custom code migration during the transition to SAP S/4HANA. Unlike new code generation, migration requires preserving functional equivalence with the source code while adapting to the constraints of the target platform — a requirement absent from existing multi-agent frameworks (MetaGPT, ChatDev, AutoGen). In this paper, the migration process is formalized as a Markov decision process, and a specialized multi-agent architecture is proposed, in which equivalence preservation is enforced through a RAG-informed Planner Agent, a fine-tuned Developer Agent, and a Reviewer Agent that uses the Integral ABAP Quality Score (IAQS) as a formalized online stopping criterion. An ablation study on 300 real ABAP fragments stratified by complexity shows that the proposed system improves IAQS to 0.845 (±0.031) versus 0.718 (±0.047) for single-pass generation, with the greatest gain observed for high-complexity fragments (+0.185 IAQS).

Keywords

multi-agent system, large language models, legacy custom code migration, intelligent reengineering, SAP ABAP, RAG, code quality metric

1. Introduction

Enterprise systems do not stand still: as SAP S/4HANA replaces legacy Enterprise Resource Planning (ERP) platforms, organizations face the need to migrate millions of lines of legacy custom code accumulated over decades of operation [1].

Legacy custom code is code developed within an organization to extend the standard functionality of the SAP system, reflecting the unique business processes of the enterprise. According to SAP, a typical SAP ERP installation contains between 2 and 10 million lines of custom ABAP code [2], the manual migration of which requires thousands of person-hours. Traditional static analysis tools such as ABAP Test Cockpit (ATC) and SAP Code Inspector can identify syntactic discrepancies and detect deprecated constructs, but they cannot offer automated solutions for business logic transformation.

The application of large language models (LLMs) and multi-agent systems (MAS) to the automation of software engineering tasks has been developing intensively in recent years. The works MetaGPT [3], ChatDev [4], and AutoGen [5] have demonstrated that role-based specialization of LLM agents can substantially improve code generation quality. However, all of the aforementioned frameworks are oriented towards the task of generating new software from a description – that is, a task in which the target code is created from scratch based on a natural language specification.

Migration is not generation. In the migration setting, the source code already exists, implements specific business logic, and has undergone years of operation in a production environment. The key requirement of migration is the preservation of functional equivalence: the migrated code must exhibit identical behavior across all business scenarios while simultaneously conforming to the constraints of the new platform (deprecated tables, modified APIs, updated syntactic rules). This

* CMIS-2026: The Ninth International Workshop on Computer Modeling and Intelligent Systems, May 5, 2026, Zaporizhzhia, Ukraine

1,* Corresponding author.

† These authors contributed equally.

oleg.pozdnyakov.ua@gmail.com (O. Pozdnyakov); parkhomenko.anzhelika@gmail.com (A. Parkhomenko)

009-0006-3955-802X (O. Pozdnyakov); 0000-0002-6008-1610 (A. Parkhomenko)

requirement is absent in the from-scratch code generation task, which gives rise to unique challenges:

- An agent cannot freely choose architectural solutions – those are constrained by the semantics of the source code.
- Quality assessment requires a formal equivalence metric rather than a subjective evaluation of " whether the code simply works".
- Migration planning requires domain knowledge specific to the target platform (SAP Notes, SAP Simplification List) that is not contained in pre-trained LLMs.

In prior works, the selection of the Qwen 2.5 Coder model [6] was substantiated, and a hybrid approach utilizing RAG (Retrieval-Augmented Generation) [7] was proposed. In addition, the Integral ABAP Quality Score (IAQS) metric [8] was developed, enabling quantitative assessment of migration output.

However, as research demonstrates [9], applying "monolithic" LLMs in zero-shot or few-shot prompting mode to migration tasks leads to error accumulation, since a single model is forced to simultaneously address heterogeneous subtasks: understanding the context of the source code, retrieving domain knowledge, generating the target code, and verifying functional equivalence.

The goal of this study is to improve the quality of automated legacy custom code migration through the application of multi-agent LLM architectures. To achieve this goal, the following objectives are addressed: to formalize the legacy custom code migration process as a Markov decision process with a functional-equivalence-aware reward function; to develop a multi-agent architecture comprising a RAG-informed Planner Agent, a fine-tuned Developer Agent, and a Reviewer Agent; to investigate the contribution of each system component through an ablation study on a real industrial corpus using the IAQS metric.

2. Related works

Large language models pre-trained on software code corpora have demonstrated significant progress in code generation, completion, and refactoring tasks. The Codex model, underpinning GitHub Copilot, was among the first commercially successful LLMs for coding, marking the beginning of intensive development of code generation models [10]. Subsequent models – StarCoder [11], the Llama family [12], and the Qwen 2.5 Coder series [13] - extended these capabilities to support a broad spectrum of programming languages, including niche enterprise languages.

However, the application of LLMs to legacy custom code migration tasks differs substantially from new code generation. In the migration setting, an LLM must simultaneously: comprehend the semantics of the source custom code, which is often written in a non-standard style employing deprecated constructs; generate functionally equivalent code for the target platform; and verify conformance with platform-specific constraints. As demonstrated in [9], such multi-tasking within a single prompt leads to quality degradation, particularly for code fragments with high cyclomatic complexity.

In prior works [6], the selection of Qwen 2.5 Coder 14B was substantiated as an optimal trade-off between ABAP code generation quality and computational requirements. A hybrid RAG-based approach (Retrieval-Augmented Generation) [7] was also proposed, enabling integration of an up-to-date knowledge base of SAP Notes and the SAP Simplification List into the generation pipeline. Nevertheless, both approaches treated migration as a single-model task — an assumption that proved insufficient for complex fragments [9].

The multi-agent systems paradigm [14] is being actively developed in the context of LLMs. Some works [3, 4, 5] have demonstrated that role-based specialization of agents can substantially improve the quality of complex task resolution. MetaGPT [3] implements the "Software Company as Multi-Agent System" paradigm, in which agents emulate the roles of product manager, architect, engineer,

and QA tester. The authors show that standardized artifacts (SOPs – Standard Operating Procedures) passed between agents significantly reduce the likelihood of hallucinations compared to unstructured interaction. However, MetaGPT is oriented towards the creation of new software from a textual description and contains no mechanisms for comparison against existing code.

ChatDev [4] models the software development lifecycle as a sequence of phases – design, coding, testing, and documentation – each governed by a pair of interacting agents operating in dialogue mode. The authors demonstrate that this approach is capable of generating working software projects within minutes. Like MetaGPT, ChatDev contains no notion of "source code" that must be preserved.

Microsoft AutoGen [5] proposes a framework for constructing arbitrary multi-agent configurations that supports human-in-the-loop operation. The system supports dynamic agent orchestration and integration with external tools. AutoGen is the most flexible of the frameworks considered. However, it provides no ready-made solutions for migration tasks, leaving the design of agents and interaction protocols entirely to the developer.

In the area of refactoring, noteworthy is the work presented in [15], which investigates the application of multi-agent LLM systems to Python code refactoring in machine learning projects. The authors show that separating responsibilities between analysis and refactoring agents improves output quality. However, this work is limited to the Python context, does not employ a domain knowledge base, and does not formalize a criterion of functional equivalence.

The migration task differs fundamentally from code generation in that it requires transforming existing code while preserving its behavior. This distinction necessitates: domain-specific planning based on external knowledge bases (SAP Notes); a formalized metric of functional equivalence; and an iterative validation cycle with a quantitative stopping criterion. The present work addresses this gap directly.

Commercial multi-agent development tools merit separate consideration. Claude Code [16] implements agent team orchestration with a lead agent/teammates pattern for parallel task execution. Google Antigravity [17] introduces an agent-first Integrated Development Environment (IDE) in which autonomous agents powered by Gemini 3 perform planning, coding, testing, and verification. However, these tools present two fundamental limitations in the context of legacy custom code migration. First, both are general-purpose development instruments and contain no mechanisms for domain-specific migration planning (interaction with SAP Notes and SAP Simplification List knowledge bases), no formalized metric of functional equivalence, and no quantitative stopping criterion for the iterative cycle. Second, Claude Code and Antigravity operate exclusively with the proprietary models of their respective developers (Claude and Gemini) via cloud APIs, rendering them inapplicable in environments subject to data transfer restrictions – in particular, SAP licensing agreements prohibit the transmission of custom ABAP code to third-party cloud services without specific authorization [1]. The system proposed in this work is deployed on-premise, thereby ensuring compliance with the information security requirements of enterprise customers.

1 summarizes the frameworks reviewed above. As the comparison shows, none of the existing systems simultaneously addresses all three requirements specific to the migration task: domain-specific planning, a formalized functional equivalence metric, and on-premise deployment. The present work targets precisely this gap.

Classical multi-agent systems theory [14] defines an agent as an entity possessing the properties of autonomy, reactivity, pro-activity, and social ability. The formalization of agent interaction has traditionally been carried out through game theory [18], Markov decision processes (MDPs) [19], and BDI (Belief-Desire-Intention) models [20]. In the context of LLM-based agents, the taxonomy of multi-agent systems for software engineering is proposed in the work [9], and three primary interaction patterns are identified: sequential (pipeline), parallel (debate), and hierarchical.

Table 1
Comparison of multi-agent frameworks for legacy custom ABAP code migration

Criterion	MetaGPT [3]	ChatDev [4]	AutoGen [5]	Claude Code [16]	Antigravity [17]	This work
Primary task	Code generation	Code generation	General-purpose	General-purpose	General-purpose	Code migration
Domain planning (SAP Notes)	-	-	-	-	-	+
Functional equivalence metric	-	-	-	-	-	+ (IAQS)
Iterative validation cycle	-	Partial	+	+	+	+
On-premise deployment	+	+	+	-	-	+
ABAP support	-	-	-	Partial	Partial	+ (fine-tuned)

The system proposed in this work implements a combination of the sequential pattern with a feedback loop, which is naturally formalized as an MDP with an iterative policy. The IAQS code quality metric employed in this work was substantiated in detail in a prior publication [8] and conforms to the principles of the ISO/IEC 25010:2023 standard [21] with respect to software product quality evaluation.

3. Materials and methods

3.1. Mathematical formalization

Before proceeding to the formalization, it is necessary to clearly establish the distinction between the task of new code generation and the task of legacy custom code migration, as this distinction determines the structure of the objective function and the state space. The code generation task is formulated as follows: given a textual description D, find a code s^* satisfying the specification [10]:

$$s^* = \arg\max_{s \in S} P(s \text{ satisfies } D). \tag{1}$$

In this task, there is no source code; quality is assessed by conformance to the description, and the agent is free to choose architectural solutions.

The migration task is formulated as follows: given source code s_0 on the platform P_{src}, find code s^* on the platform P_{tgt} that is functionally equivalent to s_0 and satisfies the constraints of P_{tgt}. In this case, the code s^* is formally defined as:

$$s^* = \arg\max_{s \in S_{tgt}} F_{quality}(s) \text{ subject to } \text{Equiv}(s, s_0) \geq \theta, \tag{2}$$

where $\text{Equiv}(s, s_0)$ is a measure of functional equivalence; θ is the equivalence threshold, and S_{tgt} is the space of syntactically correct programs on the target platform.

The presence of the equivalence constraint $\mathrm{Equiv}(s, s_0) \geq \theta$ is the key distinguishing factor that determines the structure of the entire system: the Planner Agent must identify what precisely in the code requires modification (rather than generating code anew); the Reviewer Agent must verify the preservation of behavior, and not merely correct execution.

The migration process of a software module is represented as a Markov decision process (MDP) [19], defined by the tuple:

$$M = \langle S, A, P, R, \gamma \rangle, \tag{3}$$

where S is the set of all possible migration states; A is the set of agent operations; P is the probabilistic state transition function; R is the migration quality gain function; and $\gamma \in (0,1)$ is a temporal discount coefficient penalizing solutions requiring more iterations. The state is characterized by a feature vector:

$$V(s_t) = \langle M_{syn}(s_t), M_{\text{func}}(s_t), M_{sem}(s_t), C(s_t), H(s_t) \rangle, \tag{4}$$

where $M_{syn}(s_t) \in [0,1]$ is the syntactic correctness (the proportion of lines containing no syntax errors as determined by abaplint analysis); $M_{\text{func}}(s_t) \in [0,1]$ is the functional correctness (Pass@1 on a set of ABAP Unit unit tests implementing the business scenarios of the source code):

$$M_{\text{func}}(s_t) = \frac{|test_i : test_i(s_t) = \text{pass}|}{|test_i|}; \tag{5}$$

$M_{sem}(s_t) \in [0,1]$ is the semantic similarity, computed as the cosine similarity of the embeddings of the source and current code produced by the CodeBERT model [22]:

$$M_{sem}(s_t) = \frac{\text{Emb}(s_0) \cdot \text{Emb}(s_t)}{\|\text{Emb}(s_0)\| \cdot \|\text{Emb}(s_t)\|}; \tag{6}$$

$C(s_t) \in \mathbb{N}$ is the cyclomatic complexity of the code; $H(s_t)$ is a vector of the history of preceding actions (feedback from previous iterations).

The initial state s_0 is the source custom code of SAP ERP 6.0. The target set $S^* \subset S$ is the set of states satisfying the SAP S/4HANA compatibility criterion. The Markov property is ensured by the fact that the agent's context at step t is fully determined by the current state s_t (including the history $H(s_t)$, which is explicitly incorporated into the state description).

The action space A is the set of actions available to the system:

$$A = A_{plan} \cup A_{code} \cup A_{review}, \tag{7}$$

where $A_{plan} = \{analyze, retrieve, plan\}$ is the operation set of the Planner Agent; $A_{code} = \{generate, refine\}$ is the operation set of the Developer Agent; and $A_{review} = \{validate, accept, reject\}$ is the set of actions of the Reviewer Agent.

Transition function $P : S \times A \to \Delta(S)$ [19]. Owing to the stochastic nature of LLMs, the transition between states is probabilistic:

$$P(s_{t+1} | s_t, a_t) = Pr[s_{t+1} / s_t, a_t]. \tag{8}$$

The probability distribution is determined by model parameters: generation temperature (Temp), nucleus sampling parameter (top_p), and random seed.

Reward function $R: S \times A \times S \to \mathbb{R}$ [9]. Unlike the code generation task, where the reward evaluates only "whether the code works", the reward function for the migration task must account for the preservation of equivalence:

$$R(s_t, a_t, s_{t+1}) = \Delta F_{quality}(s_t, s_{t+1}) + R_{bonus}(s_{t+1}), \tag{9}$$

where $\Delta F_{quality}(s_t, s_{t+1}) = F_{quality}(s_{t+1}) - F_{quality}(s_t)$ is a reward for quality improvement; $R_{bonus}(s_{t+1})$ is a bonus for reaching the target quality threshold ($\beta = 0.5$, $\delta = 0.76$):

$$R_{bonus}(s_{t+1}) = \begin{cases} \beta, & \text{if } F_{quality}(s_{t+1}) \geq \delta \\ 0, & \text{otherwise} \end{cases}. \tag{10}$$

The IAQS (Integral ABAP Quality Score) metric, substantiated in detail in the work, is employed as the quality evaluation function for the migrated custom code [8]:

$$F_{quality}(s) = w_1 \cdot M_{syn}(s) + w_2 \cdot M_{\text{func}}(s) + w_3 \cdot M_{sem}(s), \tag{11}$$

where $\sum_{i=1}^{3} w_i = 1$.

Based on expert assessments and preliminary experimental results [8]: $w_1 = 0.3$ (syntactic correctness), $w_2 = 0.4$ (functional correctness), $w_3 = 0.3$ (semantic similarity). The higher weight w_2 is due to the fact that functional equivalence is the critical requirement when migrating the business logic of custom code.

In the prior work [8], IAQS was used exclusively for post-hoc evaluation of migration results. In the present work, the metric is applied for the first time as an online stopping criterion for the iterative multi-agent cycle: the condition $F_{quality}(s'_{t+1}) \geq \delta$ determines whether the Reviewer Agent accepts the result or returns the code for further revision.

Thus, IAQS transitions from the role of an evaluation instrument to that of a control parameter of the system. The multi-agent system seeks a policy π^*: S ⟶ A yielding maximum expected cumulative discounted reward [19]:

$$\pi^* = \arg\max_{\pi} E\left[\sum_{t=0}^{T} \gamma^t R(s_t, \pi(s_t), s_{t+1})\right], \tag{12}$$

where $T = 3$ is the maximum number of iterations.

In this work, a greedy deterministic policy is used, defined by the agent architecture (Section 4): a fixed sequence Plan ⟶ Code ⟶ Review with an iterative cycle when $F_{quality} < \delta$. The MDP formalization is not used in the current implementation for policy learning – it serves two purposes: it provides a rigorous mathematical description of the migration process, and it creates a foundation for transitioning to a learnable policy based on reinforcement learning methods.

3.2. Multi-agent system architecture

The proposed system implements a multi-agent architecture (1), in which each agent satisfies the classical agency criteria formulated in [14].

Each agent operates on the basis of its own LLM instance with a unique system context (System Prompt) and tool set. Agents make decisions autonomously, without direct control from other agents: the Planner Agent independently determines the analysis strategy and retrieves relevant SAP Notes; the Developer Agent decides how to implement the plan without waiting for instructions; the Reviewer Agent evaluates the result and formulates structured feedback on its own. This distinguishes the proposed system from a simple pipeline, in which each step is deterministically

defined by its input. Agents react to changes in the environment: the Reviewer Agent initiates a refinement cycle upon detecting $F_{quality} < \delta$; the Developer Agent adapts its generation strategy upon receiving feedback, correcting specific code fragments instead of complete regeneration.

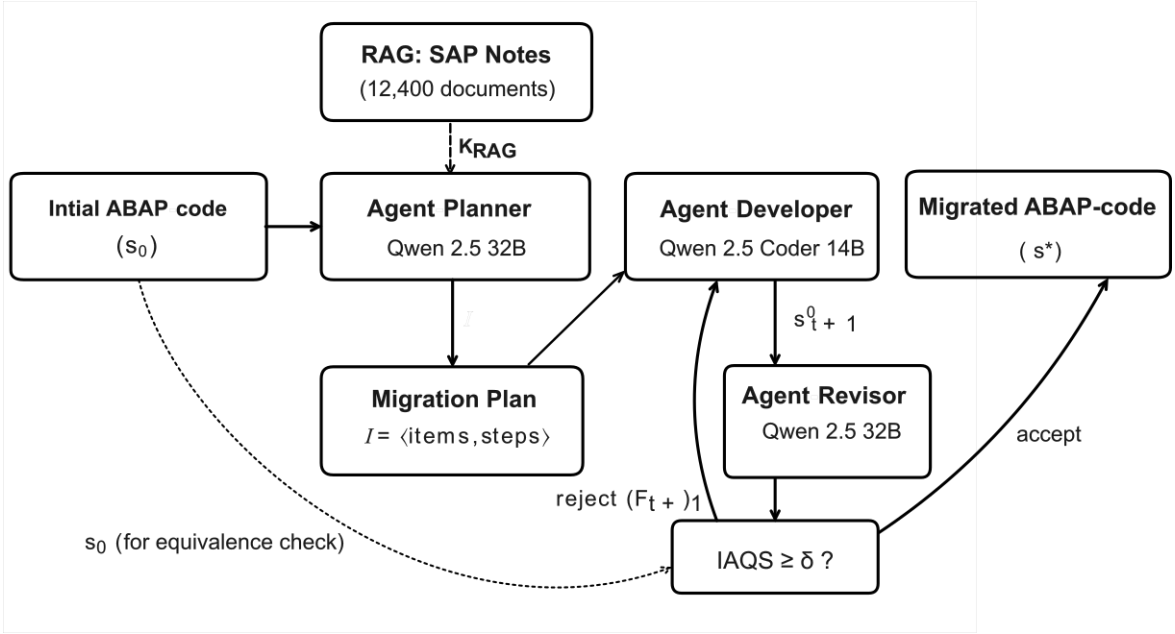

Figure 1: Architecture of the multi-agent system for legacy custom code migration.

Agents exhibit goal-directed behavior: the Planner Agent proactively identifies potential migration problems (typing conflicts, implicit dependencies) before code generation begins; the Reviewer Agent performs semantic checks when edge cases are detected. Agents interact through a formalized protocol for exchanging structured messages. Unlike isolated pipeline components, agents exchange semantically rich artifacts: a migration plan with justification, generated code with comments and feedback with classified remarks and recommendations.

2 shows the agent interaction protocol with the iterative validation cycle.

Step 1. Planning. The Planner Agent receives the source custom code s_0 and produces a migration plan. The planning procedure is formally expressed as:

$$I = a_{plan}\left(s_0, K_{RAG}\right) \in M_{msg}. \tag{13}$$

Step 2. Generation. The Developer Agent implements the plan. The code generation step is formally described as:

$$s'_{t+1} = a_{code}\left(s_t, I, F_t\right) \in S, \tag{14}$$

where F_t is the feedback from the Reviewer Agent ($F_0 = \varnothing$ at the first iteration).

Step 3. Validation. The Reviewer Agent evaluates the result. The validation output is formally represented as:

$$\left\langle d_t, F_{t+1} \right\rangle = a_{review}\left(s_0, s_t, s'_{t+1}\right), \tag{15}$$

where $d_t \in accept, reject$. Critically important is the fact that the Reviewer Agent receives the source code s_0 as input – this ensures verification of functional equivalence.

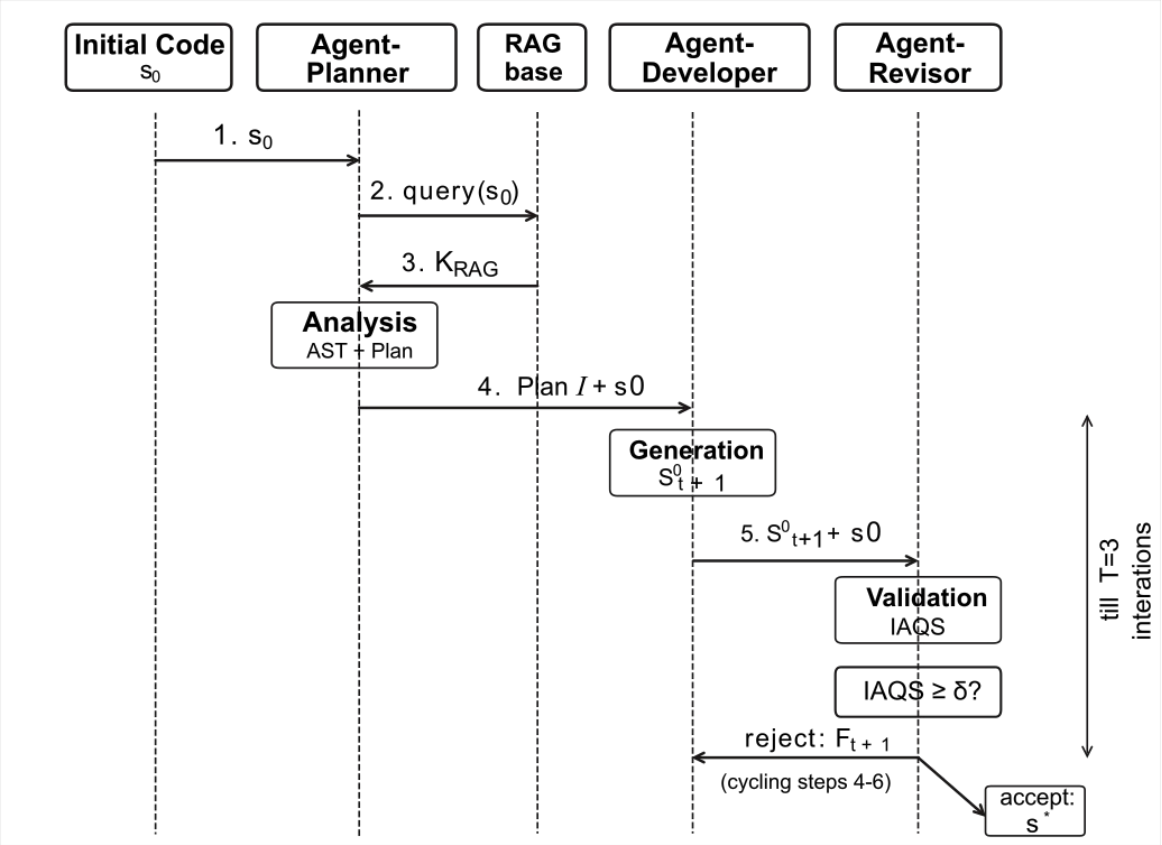

Figure 2: Agent interaction protocol with iterative validation cycle.

Step 4. Decision making is formally defined as:

$$s_{t+1} = \begin{cases} s'_{t+1}, & \text{if } d_t = accept \text{ or } t = T \\ s_t, & \text{for repeated iteration (go to Step 2)} \end{cases}. \quad (16)$$

Acceptance condition: $d_t = accept \Leftrightarrow F_{quality}\left(s'_{t+1}\right) \geq \delta$, where $\delta = 0.76$. The threshold $\delta = 0.76$ was selected based on the minimum acceptable quality level established empirically in prior work [8].

Model: Qwen 2.5 32B Instruct (a mid-size model providing sufficient reasoning quality when deployed on a single GPU). For analysis and planning tasks that do not require code generation, the 32B model demonstrates quality comparable to full-size 72B variants [13] at half the GPU memory requirements. Temperature: Temp = 0.4. Context: the System Prompt contains the role of "SAP Migration Architect" and instructions for ABAP code analysis with reference to the SAP S/4HANA Simplification List.

The task of this agent is the semantic analysis of the source custom code and the formation of a domain-informed migration plan. The agent uses the RAG mechanism to access the external knowledge base K_{RAG}containing 12,400 SAP Notes documents and 2,300 SAP Simplification List entries [7].

Agent function:

$$I = a_{plan}\left(s_0, K_{RAG}\right) = \langle \text{deprecated\_items}, \text{replacements}, \text{steps}, \text{risks} \rangle. \quad (17)$$

The agent analyses the AST (Abstract Syntax Tree) of the source code $s_0$using abaplint, identifies deprecated constructs, and produces a step-by-step refactoring plan. Each step of the plan contains:

a reference to a specific code element (with line number), the type of change required, a reference to the corresponding SAP Note, and a risk warning.

It is the work of the Planner Agent that provides the domain specificity of the system, absent in generic frameworks. Without knowledge of SAP Notes and SAP Simplification List, the agent cannot correctly determine which constructs are deprecated and what they should be replaced with.

Model Qwen 2.5 Coder 14B, fine-tuned using QLoRA [23] on 5,000 "legacy ABAP code → modern ABAP code" pairs [6]. Temperature: $Temp = 0.2$; $\text{top}_p = 0.9$; max_tokens= 4096. Context: the System Prompt contains the role of "Senior ABAP Developer".

State transition function:

$$s'_{t+1} = a_{code}(s_t, I, F_t). \tag{18}$$

During the first iteration ($t = 0$), the agent receives the source code $s_0$and the plan I. In subsequent iterations, feedback F_t is added to the context. The agent performs incremental refinement – it corrects the indicated problems rather than regenerating the code entirely.

Model Qwen 2.5 32B Instruct. As with the Planner Agent, the Reviewer Agent's task is analysis and evaluation rather than code generation, which allows a mid-size model to be used without significant loss of validation quality. Temperature: $Temp = 0.1$. Context: the System Prompt contains the role of "Code Quality Auditor".

The agent implements two-stage validation:

Stage 1. Automatic verification. Computation of metrics M_{syn} (abaplint v2.x), M_{func} (ABAP Unit), M_{sem} (CodeBERT [22] cosine similarity), and the aggregated $F_{quality}$.

Stage 2. LLM analysis. If $F_{quality}(s'_{t+1}) < \delta$, the agent generates structured feedback F_{t+1}: a list of issues with line numbers and severity levels, a classification (syntactic, semantic, functional), and recommendations with references to SAP Notes.

The Reviewer receives as input not only the generated code s'_{t+1} , but also the source custom code s_0, which enables verification of functional equivalence preservation.

All models are deployed on a server with 1× NVIDIA A100 40GB via the vLLM [24] inference engine (v0.4.x). The choice of the NVIDIA A100 40GB is determined by its availability within the enterprise infrastructure of the migration project and by the peak memory requirement of the sequential agent execution scheme: loading the largest model (32B in GPTQ-Int4 quantization) requires up to 36GB of GPU memory, which fits within the 40GB capacity. Upgrading to a higher-class GPU (A100 80GB or H100) would enable simultaneous loading of all agents and reduce per-fragment processing time from 95 to approximately 40 seconds; however, this is not a prerequisite for reproducing the reported results. The choice of 14B (Coder) and 32B (Planner, Reviewer) model sizes is determined by the video memory constraints of a single GPU: models are loaded sequentially (not in parallel), which allows even the 32B model in GPTQ-Int4 quantization to fit within 40GB. The Developer Agent (14B) uses a QLoRA fine-tuned version, which precludes using a larger model without complete retraining. The RAG pipeline is implemented using LangChain with a ChromaDB vector store (HNSW index, embeddings-all-MiniLM-L6-v2 [25], dimension 384). Average processing time per custom code fragment: 35 seconds for single-pass generation (Baseline-1), 95 seconds for the full three-agent cycle with 1 refinement iteration.

Sequential agent execution reflects a deployment constraint, not an architectural one. Multi-agency is determined by agent autonomy, context isolation, and a formalized interaction protocol, not by execution parallelism [14]. Each agent operates based on a separate model (or a separate fine-tuned version) with its own system context and has no access to the internal state of other agents, only to the structured messages of the protocol. When scaling to multiple GPUs, agents can execute in parallel without changing the interaction protocol. Thus, the proposed architecture is decoupled from the specific deployment infrastructure.

4. Experiments

For validation of the proposed model, a dataset consisting of 300 real legacy custom code fragments extracted from SAP ERP 6.0 customizations within a migration project in which the authors participated was used. Fragments were selected according to the following criteria: presence of at least one deprecated construct detected by ABAP Test Cockpit; length from 20 to 500 lines of code (LOC); presence of corresponding unit tests (ABAP Unit) confirming the functional correctness of the source code.

Distribution of fragments by complexity: 102 low-complexity fragments (1–2 deprecated constructs, cyclomatic complexity ≤ 10), 126 medium-complexity (3–5 deprecated constructs, cyclomatic complexity 11–25), 72 high-complexity (> 5 deprecated constructs, cyclomatic complexity > 25).

To evaluate the contribution of each system component, four configurations were compared:

1. Baseline-1 (Zero-shot). The Baseline-1 configuration employs the QLoRA fine-tuned Qwen 2.5 Coder 14B model, reported in [8] (IAQS = 0.718), without the ORPO (Odds Ratio Preference Optimization) alignment stage, as the reference point for measuring the isolated contribution of the multi-agent architecture.
2. Baseline-2 (CoT). The same model employing Chain-of-Thought prompting [26]: firstly, the model is prompted to enumerate the migration steps, followed by a prompt to generate the code.
3. Baseline-3 (Dual-Agent). A two-agent configuration: Coder (Qwen 2.5 Coder 14B) and Reviewer (Qwen 2.5 32B Instruct) with an iterative cycle up to $T = 3$. Planning functions are assigned to the Developer Agent.
4. MAS (Proposed). The full three-agent system: Planner (Qwen 2.5 32B Instruct) + Coder (Qwen 2.5 Coder 14B) + Reviewer (Qwen 2.5 32B Instruct) with an iterative cycle up to $T = 3$.

Each experiment was repeated 3 times with different random seeds to assess the result variance. Statistical significance of differences was evaluated using the paired Student's t-test with significance level $\alpha = 0.05$. The comparison of approach effectiveness is presented in 2.

All differences between the MAS and each of the baseline configurations are statistically significant ($p < 0.01$). 3 illustrates the IAQS comparison across all four configurations.

Table 2
Comparison of approach effectiveness (mean ± standard deviation, 3 runs)

Method	$1 - M_{syn}$(errors, ↓)	M_{sem} (↑)	M_{func} (Pass@1, ↑)	IAQS (↑)
Baseline-1 (Zero-shot)	15.4% ± 2.1%	0.761 ± 0.038	0.642 ± 0.045	0.718 ± 0.047
Baseline-2 (CoT)	11.8% ± 1.8%	0.794 ± 0.032	0.689 ± 0.041	0.762 ± 0.039
Baseline-3(Dual-Agent)	7.1% ± 1.5%	0.851 ± 0.029	0.761 ± 0.037	0.806 ± 0.035
MAS (Proposed)	4.2% ± 1.1%	0.892±0.025	0.803±0.031	0.845±0.031

Each architectural addition contributes measurably: CoT reasoning over Zero-shot (+0.044 IAQS), iterative feedback over CoT (+0.044), and the dedicated Planner Agent over the two-agent baseline (+0.039) — the latter being the most structurally significant addition.

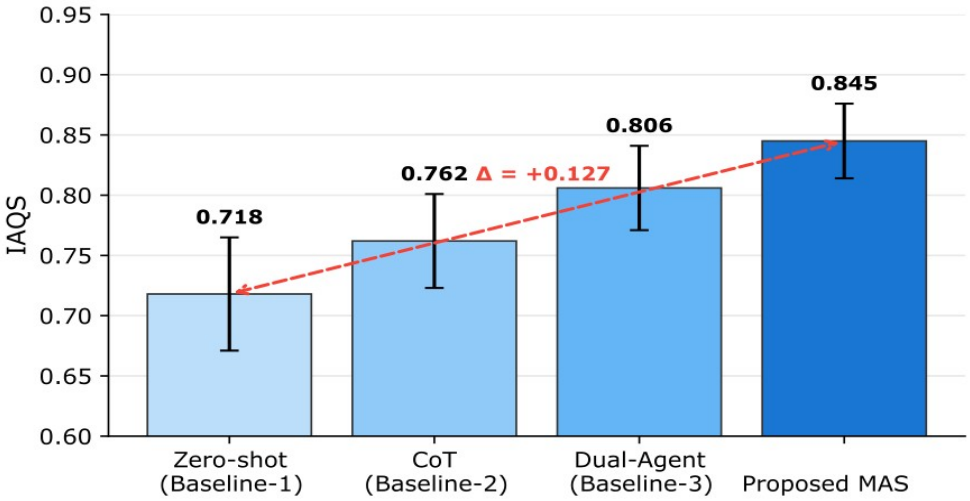

Figure 3: Comparison of the integral quality score IAQS for four configurations.

3 presents IAQS scores across complexity levels.

Table 3
IAQS by complexity levels of custom code fragments

Method	Low (n=102)	Medium (n=126)	High (n=72)
Baseline-1 (Zero-shot)	0.812 ± 0.035	0.711 ± 0.042	0.587 ± 0.058
Baseline-2 (CoT)	0.841 ± 0.030	0.754 ± 0.037	0.649 ± 0.051
Baseline-3 (Dual-Agent)	0.869 ± 0.027	0.802 ± 0.033	0.718 ± 0.044
MAS (Proposed)	0.895 ± 0.022	0.847 ± 0.028	0.772 ± 0.039
Δ (MAS vs Zero-shot)	+0.083	+0.136	+0.185

4 illustrates the distribution of IAQS scores across complexity levels of the code fragments. The advantage of the MAS is most pronounced for high-complexity fragments: the difference relative to Zero-shot amounts to +0.185, compared to +0.083 for low-complexity fragments. The gap between Dual-Agent and MAS increases with complexity: +0.026 for low, +0.045 for medium, and +0.054 for high complexity. This indicates that the contribution of the Planner Agent is proportional to the complexity of the task.

The effect of the number of iterations is shown in 4.

Table 4
Effect of the number of iterations on IAQS (full MAS system)

Iteration count	IAQS	M_{syn}	M_{func}	Rejection rate
1(without refinement)	0.798 ± 0.034	0.916 ± 0.022	0.752 ± 0.035	-
2	0.833 ± 0.031	0.948 ± 0.018	0.789 ± 0.032	42%
3	0.845 ± 0.031	0.958 ± 0.015	0.803 ± 0.031	11%

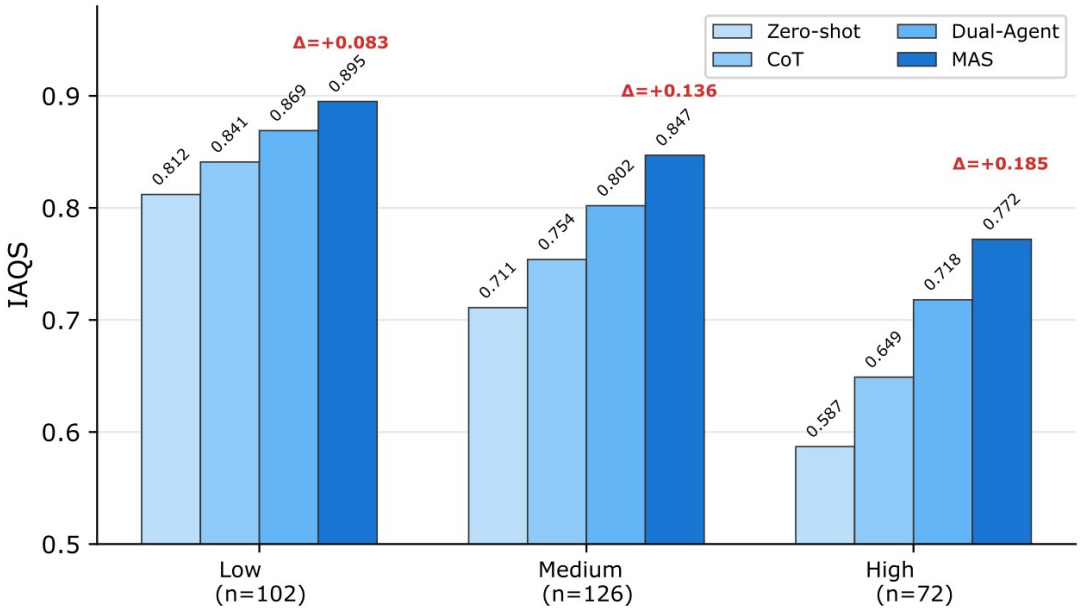

Figure 4: IAQS by complexity levels of custom code fragments.

The greatest improvement is observed in the transition from 1 to 2 iterations (+0.035 IAQS). The third iteration provides a smaller but statistically significant contribution (+0.012, $p < 0.05$). Log analysis shows: 42% of refinement cases on the first iteration, 11% on the second, 3% on the third. Rapid convergence (three iterations sufficient for 97% of fragments) confirms the practical applicability of the approach.

The error distribution by type is summarized in 5.

Table 5
Error distribution by type (total across 300 fragments, mean over 3 runs)

Error type	Zero-shot	CoT	Dual-Agent	MAS
Incorrect method signature	26	19	9	5
Deprecated construct in output code	16	12	7	3
Type violation	13	11	6	3
Functional mismatch	34	27	19	14
Other	11	9	5	4
Total	100	78	46	29

The greatest reduction is observed in the "Incorrect method signature" category (26 → 5, a 5.2-fold decrease). This error type is attributable to the fact that, during migration to S/4HANA, many function modules and BAPIs are replaced by new ones with modified signatures. The Planner Agent pre-retrieves the correct signatures from the SAP Notes RAG knowledge base.

The "Deprecated construct in output code" category (16 → 3, a 5.3-fold decrease) also demonstrates substantial improvement: an explicit migration plan containing a list of constructs subject to replacement minimizes the likelihood of their reproduction.

The "Functional mismatch" category (34 → 14) exhibits the smallest relative reduction, indicating that ensuring full functional equivalence remains the most challenging aspect of migration.

Functional mismatches are attributable to the fundamental difficulty of the equivalence verification problem. Unlike syntactic errors, which are deterministically detected by static analysis,

functional equivalence can be violated in edge-case scenarios not covered by the available unit test suite.

5. Results and discussion

The experiments were conducted on a corpus of 300 real custom ABAP code fragments extracted from a real migration project. However, the results are subject to limitations of external validity related to the size and origin of the corpus. Within these constraints, the following results were obtained:

1. The legacy custom code migration process is formalized as a Markov decision process with a functional-equivalence-aware reward function, thereby demonstrating a key distinction from existing code-generation formulations.
2. A three-agent architecture satisfying the Wooldridge–Jennings agency criteria is developed, with a formalized structured message exchange protocol.
3. Experimental validation on 300 real ABAP fragments confirmed the effectiveness of the approach: IAQS increased from 0.718 ± 0.047 to 0.845 ± 0.031, and syntactic errors decreased from 15.4% to 4.2% (see Table 1).
4. Ablation experiments showed that the Planner Agent's contribution is proportional to code complexity (+0.054 IAQS for high complexity vs. +0.026 for low, see Table 2); the iterative validation cycle ensures rapid convergence — 97% of fragments are resolved in ≤ 3 iterations (see Table 3).
5. The MDP formalization provides a foundation for transitioning to an RLHF-based learnable agent orchestration policy.

Notably, the baseline single-pass approach (IAQS = 0.718) falls below the acceptance threshold $\delta = 0.76$, whereas the proposed MAS achieves IAQS = 0.845, exceeding the threshold by 0.085. This confirms that the multi-agent architecture is not merely an incremental improvement — without it, the system fails to meet the minimum quality criterion established in prior work [8].

The obtained results support the hypothesis that the separation of responsibilities between specialized agents improves the quality of automated code migration. The most significant improvement is observed when transitioning from monolithic generation to a multi-agent architecture with iterative feedback. In particular, the introduction of a dedicated Planner Agent that utilizes a domain knowledge base composed of SAP Notes and the SAP Simplification List allows the system to identify constructs requiring adaptation during the transition to SAP S/4HANA in advance. This substantially reduces the likelihood of generating obsolete constructs and incorrect method signatures in the resulting code.

The analysis across complexity levels shows that the benefits of the multi-agent architecture increase with the structural complexity of the migrated fragments. For high-complexity fragments, the difference between the MAS-based approach and single-pass generation reaches +0.185 in IAQS. This indicates that explicit planning and iterative validation become particularly important when migrating code containing multiple dependencies and legacy constructs.

The influence of the iterative validation cycle is also noteworthy. Experimental results show that the largest quality improvement occurs after the second iteration, while subsequent iterations contribute significantly less to overall quality gains. This behavior suggests a rapid convergence of the error-correction process and confirms the practical applicability of the proposed architecture for processing large corpora of custom code.

Despite the observed improvements, the category of errors related to functional inconsistencies shows the smallest relative reduction among error types. Even when the full MAS configuration is used, 14 out of 300 fragments still demonstrate behavioral discrepancies compared to the original code as detected by the available unit tests in the evaluation corpus. These cases represent only a subset of all failures reflected in the Pass@1 metric, which also accounts for compilation errors and

other migration defects. Functional mismatches are attributable to the fundamental difficulty of the equivalence verification problem: unlike syntactic errors, which are deterministically detected by static analysis, functional equivalence can be violated in edge-case scenarios not covered by the available unit test suite. Consequently, the M_{func} metric is bounded by the completeness of the test coverage of the source code — a fundamental limitation of the evaluation methodology rather than an architectural deficiency of the proposed system.

The gains come at a cost: the full MAS pipeline requires approximately 2.5–3× as much compute as monolithic generation due to sequential LLM calls per fragment. At the scale of a real migration project — hundreds of thousands of custom code fragments — this translates to a need for scalable infrastructure and parallel agent execution across multiple GPUs.

A further limitation concerns the domain specificity of the proposed architecture. The system is designed to migrate custom ABAP code during the transition from SAP ERP 6.0 to SAP S/4HANA and relies on a specialized knowledge base derived from SAP Notes and the SAP Simplification List.

Nevertheless, the proposed architecture is modular by design. The RAG-informed Planner Agent can be reconfigured to operate with knowledge bases of other enterprise platforms, such as Oracle E-Business Suite or Microsoft Dynamics, by replacing the domain-specific document corpus while keeping the agent interaction protocol unchanged. Similarly, the IAQS metric generalizes to any programming language for which a static analyzer and a unit test suite are available. Thus, the core architectural principles of the proposed system are transferable to code migration tasks beyond the ABAP/SAP domain, provided that an appropriate domain knowledge base is assembled.

6. Conclusions

The paper proposes and formalizes a multi-agent architecture for intelligent reengineering of legacy software, specialized for the legacy custom code migration task. Unlike existing multi-agent frameworks (MetaGPT, ChatDev, AutoGen) and commercial tools (Claude Code, Google Antigravity) oriented towards new code generation, the proposed system explicitly accounts for the key requirement of migration – the preservation of functional equivalence with the source code.

The scientific novelty of the work lies in the further development of the multi-agent architecture of the reengineering system through the formalization of the custom code migration task as a Markov Decision Process with a reward function based on the IAQS metric, which, unlike existing approaches, provides a formalized online stopping criterion for the iterative cycle between software agents — the planner, developer, and reviewer.

The practical significance of this work lies in the fact that the proposed multi-agent system architecture enables automated migration of custom ABAP code, resulting in a 17.7% improvement in the integrated quality metric compared to single-pass generation. The system is deployed on readily available hardware (1× NVIDIA A100 40 GB). It is compatible with SAP licensing restrictions, as it does not require migrating legacy custom code to third-party cloud services.

Thus, the present work demonstrates that legacy custom code migration is a fundamentally distinct problem from code generation and requires a specialized architecture with explicit functional equivalence control. The proposed system surpasses the minimum acceptance threshold $\delta=0.76$ — unachievable by single-pass approaches — confirming the practical viability of the multi-agent architecture for industrial-scale SAP S/4HANA migration projects.

Future work directions include: transition from a greedy policy to a learnable one based on RLHF; parallel agent execution across multiple GPUs to reduce per-fragment processing time from 95 to ~40 seconds; dynamic agent orchestration with adaptation to fragment complexity; scaling to other enterprise platforms; and integration with the SAP continuous integration pipeline.

Declaration on Generative AI

The Qwen 2.5 Coder 14B and Qwen 2.5 32B Instruct models were utilized in the "Materials and Methods" and "Experiment" sections as the core components of the proposed multi-agent

architecture. Their outputs were verified using static analysis (abaplint), unit tests (ABAP Unit), and the IAQS metric, and they served as the basis for the experimental data presented in the paper.

Additionally, during the preparation of this paper, the authors used Claude check grammar and spelling, paraphrase, and reword. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] P. Hardy, Migrating Custom Code to SAP S/4HANA, Rheinwerk Publishing, Boston, MA, 2020.

[2] M. Mergaerts, B. Vanstechelman, SAP S/4HANA System Conversion Guide, 1st ed., SAP PRESS, Rheinwerk, 2020.

[3] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, J. Schmidhuber, MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework, in Proceedings of the International Conference on Learning Representations (ICLR 2024), Vienna, Austria, 2024. URL: https://proceedings.iclr.cc/paper/2024/hash/6507b115562bb0a305f1958ccc87355a-Abstract.html.

[4] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, M. Sun, ChatDev: Communicative Agents for Software Development, in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL 2024), Bangkok, Thailand, 2024, pp. 15174–15186. doi:10.18653/v1/2024.acl-long.810.

[5] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. Awadall, R. White, D. Burger, C. Wang, AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation, in Proceedings of the Conference on Language Modeling (COLM 2024), Philadelphia, PA, 2024. URL: https://openreview.net/forum?id=BAakY1hNKS.

[6] O. Pozdnyakov, A. Parkhomenko, Research and Selection of Large Learning Models for Automation of ABAP-Code Migration, Management of Development of Complex Systems 63 (2025) 191–200.

[7] O. Pozdnyakov, A. Parkhomenko, Hybrid Approach to SAP ABAP Custom Code Migration, in Proceedings of the Second International Scientific and Practical Conference “Artificial Intelligence and Information Technologies” (AIT-2025), Kyiv, Ukraine, 2025, pp. 70–71.

[8] O. Pozdnyakov, A. Parkhomenko, Evaluation and Quality Assurance of Migrated ABAP Code using an Integral Metric and Generative Artificial Intelligence Models, Radio Electronics, Computer Science, Control 1 (2026) 80–89.

[9] J. He, C. Treude, D. Lo, LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead, ACM Trans. Softw. Eng. Methodol. (TOSEM) 34 (2025) 1–30. doi:10.1145/3712003.

[10] J. Jiang, F. Wang, J. Shen, S. Kim, S. Kim, A Survey on Large Language Models for Code Generation, ACM Trans. Softw. Eng. Methodol. (TOSEM) (2024). doi:10.1145/3747588.

[11] R. Li, L. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, O. Adelani et al., StarCoder: may the source be with you!, Trans. Mach. Learn. Res. (2023). doi:10.48550/arXiv.2305.06161.

[12] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt et al., Code Llama: Open Foundation Models for Code, (2023). doi:10.48550/arXiv.2308.12950.

[13] Qwen Team. Qwen2.5-Coder: Powerful, Diverse, Practical. Qwen Blog, 2024. URL: https://qwenlm.github.io/blog/qwen2.5-coder-family/.

[14] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd ed., John Wiley & Sons, Chichester, 2009.

[15] V. Rajendran, D. Besiahgari, S. C. Patil, M. Chandrashekaraiah, A Multi-Agent LLM Environment for Software Design and Refactoring: A Conceptual Framework, in Proceedings of the 2025 IEEE SoutheastCon, Concord, NC, USA, IEEE, 2025. doi:10.1109/SoutheastCon56624.2025.10971563.

[16] Anthropic. Claude Code: Agentic Coding Tool Documentation, 2025. URL: https://code.claude.com/docs/en/overview.
[17] Google. Antigravity: Agent-First Development Platform. Google Developers Blog, 2025. URL: https://developers.googleblog.com/build-with-google-antigravity-our-new-agentic-development-platform/.
[18] Y. Shoham, K. Leyton-Brown, Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, Cambridge University Press, Cambridge, 2009.
[19] M. L. Puterman, Markov Decision Processes: Discrete Stochastic Dynamic Programming, John Wiley & Sons, Hoboken, NJ, 2005.
[20] A. S. Rao, M. P. Georgeff, BDI Agents: From Theory to Practice, in Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, CA, 1995, pp. 312–319.
[21] ISO/IEC 25010:2023, Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — Product quality model, ISO, 2023.
[22] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, in Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 2020, pp. 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139.
[23] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, QLoRA: Efficient Finetuning of Quantized LLMs, in Advances in Neural Information Processing Systems 36, New Orleans, LA, 2023. doi:10.48550/arXiv.2305.14314.
[24] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. Yu, J. Gonzalez, H. Zhang, I. Stoica, Efficient Memory Management for Large Language Model Serving with PagedAttention, in Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023), Koblenz, Germany, 2023, pp. 611–626. doi:10.1145/3600006.3613165.
[25] N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP) and the 9th International Joint Conference on Natural Language Processing (IJCNLP), Hong Kong, China, 2019, pp. 3982–3992. doi:10.18653/v1/D19-1410.
[26] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, D. Zhou, Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, in Advances in Neural Information Processing Systems 35, New Orleans, LA, 2022. doi:10.48550/arXiv.2201.11903.
[27] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, R. Lowe et al., Training Language Models to Follow Instructions with Human Feedback, in Advances in Neural Information Processing Systems 35, New Orleans, LA, 2022. doi:10.48550/arXiv.2203.02155.