Traceable Documentation-Grounded Code Generation for Proprietary Industrial Languages: A DENSO RC7 PAC Case Study

Sergio Oviedo-Seas*, Ignacio Trejos-Zelaya*, María Jesús Rodríguez-Molina† and Sharon Castro-Mata†
*School of Software Engineering, Universidad CENFOTEC, San José, Costa Rica
†Robotics Laboratory (RobLab), Universidad CENFOTEC, San José, Costa Rica
soviedo@ucenfotec.ac.cr, itrejos@ucenfotec.ac.cr, mrodriguezmo@ucenfotec.ac.cr, scastromat@ucenfotec.ac.cr

***Abstract*—Proprietary industrial robot languages are low-resource by construction: their documentation is vendor-controlled, their compilers are vendor tools, and, to the authors' knowledge, no publicly available dedicated training corpus exists. Coverage is nonetheless not zero. Publicly accessible documentation of PAC (Programming language for Assembly Cell), the language of DENSO RC7 robots in WinCaps III, provides a plausible route into model pretraining, and most current large language models (LLMs) will produce PAC-like code from it alone. What they will not do is produce it reliably: in a structured comparative evaluation, four current LLMs queried without in-context grounding compiled PAC in 1 of 35 runs in WinCaps III (2.9%), each failing in a distinct, systematic way, and none supplied a documentation citation that could be checked. An engineer cannot tell in advance which output is trustworthy for a given hardware configuration. The RC7 Programming Assistant grounds generation in documentation carried in two forms: syntax rules distilled once from the manuals, and passages retrieved per query from 2,982 pages across eleven official DENSO manuals, indexed as 3072-dimensional embeddings under a Hierarchical Navigable Small World (HNSW) index and selected using Hypothetical Document Embeddings (HyDE). It ingests installation-specific documentation—I/O tables, taught positions, program banks—that exists in no public corpus, and resolves each inline source reference through a map the model cannot fabricate. Across the three evaluated PAC tasks, under the same criterion, its final post-lint programs compiled in all nine runs, and all 44 emitted inline citations resolved to manual pages containing the cited instructions.**

***Index Terms*—Documentation grounding, Retrieval-Augmented Generation, HyDE, proprietary industrial languages, industrial robotics, PAC programming, DENSO RC7, WinCaps III, AI auditability, verified traceability, domain-specific code generation, LLM evaluation, compilation verification.**

I. Introduction

Industrial robot programming remains a significant barrier to adoption in small and medium manufacturing environments. DENSO RC7 robots, widely deployed in precision assembly applications, are programmed using PAC (Programming language for Assembly Cell), a proprietary structured language developed by DENSO and executed within the WinCaps III IDE [32]. PAC was designed to increase efficiency in the development and maintenance of robot control programs over conventional languages; it is upward compatible with SLIM (Standard Language for Industrial Manipulators), the industrial robot language standardized by the Japanese Industrial Standards (JIS) committee, supports multitasking and interruption processing, and enables control of both robot motion and vision devices within a single program [32]. PAC applies exclusively to RC7 and earlier controllers; DENSO's RC8 and later controllers use PACScript, an evolution of PAC that retains the same foundational syntax but introduces variations in reserved words, instruction set, and non-motion control structures [32]. PAC is effectively a hybrid language: its control-flow and non-motion instructions resemble Visual Basic, whereas its motion instructions conform to SLIM; PACScript shares this same dual lineage. PAC programming requires engineers to maintain awareness of robot-specific I/O configurations, variable table structures, and controller version constraints—knowledge typically dispersed across multiple technical manuals totaling thousands of pages.

A. The LLM Coverage Problem is Heterogeneity, Not Absence

A natural expectation would be to use Large Language Models (LLMs) to assist with PAC programming. The reality is more nuanced than a simple can/cannot dichotomy. Because DENSO documentation is partially available in public sources, most current LLMs can produce PAC-like code from pretraining alone. However, this coverage is heterogeneous and its reliability is not guaranteed: generated programs may contain fabricated instructions, omit configuration-specific details, or fail to compile in WinCaps III, and the engineer has no way to know in advance which output is trustworthy. To characterize this behavior systematically, this work includes a structured comparative evaluation of four current LLMs, described in Section IV.

This heterogeneity creates an operationally problematic condition. An engineer without deep PAC expertise may not be able to determine in advance which model will succeed, which instructions are fabricated, or whether a compiled program is correct for their specific RC7 hardware configuration. In safety-critical industrial environments, statistically probable correctness is not a sufficient deployment criterion—verifiability and traceability to authoritative sources are required [23], [30].

B. Documentation Grounding as the Mechanism

The limitation above is not primarily one of generative capability but of verifiable grounding: without the relevant documentation in the generation context, neither the model nor the user can guarantee that an instruction is valid for the target controller and configuration. Retrieval-Augmented Generation (RAG) [5] addresses this by retrieving relevant passages from official DENSO RC7 documentation [33] at inference time, so that generation is anchored in verified sources rather than in opaque pretraining coverage. Siriwardhana et al. [21] demonstrate that standard RAG systems trained on general corpora fail in specialized domains—motivating domain-specific pipelines such as the one presented here. Hypothetical Document Embeddings (HyDE) [6] further bridge the semantic gap between short natural language queries and technical documentation excerpts.

C. Contributions

This paper presents the RC7 Programming Assistant and makes the following contributions:

1) A structured evaluation protocol for LLM PAC generation—fixed prompts, a common compilation criterion—covering four current LLMs (ChatGPT, Gemini, Claude, Grok), recorded by model and version and queried without in-context grounding, on representative PAC tasks, with compilation in WinCaps III as the objective validation criterion and multiple runs per task to observe repeated-run consistency.
2) Empirical evidence motivating the system: without in-context grounding, current LLMs are not a dependable option for PAC—only 1 of 35 runs compiled in WinCaps III, with distinct per-model failure modes and no verifiable documentation citations.
3) A system that carries documentation into the generation context in two complementary forms: *resident*—PAC syntax rules distilled from the manuals, invariant across queries—and *retrieved*—passages selected per query by a four-phase pipeline over 2,982 pages of official documentation, with HyDE-augmented retrieval, LLM-driven chunk validation during ingestion, robot-configuration-aware filtering, and SSE streaming. The rules cover what every PAC program needs; retrieval covers what this query needs, including installation-specific material that no rule set can anticipate.
4) Traceability as a property of the pipeline rather than of model self-report: each inline source reference resolves through a map built from the retrieved chunks and persisted with the message, and the model's own citation array is discarded, so a provenance claim can be neither invented nor silently re-decoded in a later turn. Which instructions carry a reference remains the model's choice; whether the referenced page documents the instruction is what we verify. Generated code additionally passes two-level verification: deterministic linting plus read-only semantic advisories.

II. Background

A. LLMs for Code Generation in Low-Resource Languages

Code generation with LLMs has advanced rapidly for mainstream languages. Codex [10] introduced HumanEval and demonstrated strong performance in Python; Code Llama [11] extended this to open models. However, all major benchmarks target languages with substantial pretraining representation. Joel et al. [2] provide the most comprehensive survey of LLM-based code generation for low-resource programming languages (LRPLs) and domain-specific languages (DSLs), identifying the core challenge: LLMs trained predominantly on high-resource languages generalize poorly to LRPLs and DSLs, exhibiting dialect conflation, hallucinated functions, and IDE-language boundary confusion. Gu et al. [3] confirm that standard benchmarks such as HumanEval fail to characterize the challenges of domain-specific code generation. Alfonseca and Martín Colino [1] confirm the problem empirically for IBM APL2 (LRPL) and Engineering Equation Solver (DSL), showing that no state-of-the-art LLM generates compilable code in either language. PAC belongs to both families simultaneously: it is low-resource (documentation is partially available through public aggregators, but no dedicated PAC training corpus exists, and LLM coverage is empirically inconsistent across models) and domain-specific (executable only within WinCaps III, with installation-specific context requirements—I/O tables, taught positions, tool configurations—that are never publicly available [32]). The present work instantiates and validates the system on PAC for RC7. What is domain-specific is the instantiation—the indexed corpus, the resident syntax rules, and the verification rules—rather than the architecture that carries them; porting to PACScript on RC8, or to another vendor's language, means supplying those three and revalidating instruction-set coverage, not redesigning the pipeline.

B. Retrieval-Augmented Generation

RAG [5] augments LLM generation with retrieved passages from an external knowledge base, enabling responses grounded in verified documentation without retraining [12]. A persistent challenge is the semantic gap between short queries and longer document passages. Moreover, standard RAG systems trained on general-domain corpora do not adapt well to specialized domains without domain-specific tuning or retrieval grounding [21]—a limitation directly relevant to PAC, where no general-purpose pretraining data exists. HyDE [6] addresses this gap by generating a hypothetical answer to the query before retrieval, then embedding the combined (query $\|$ hypothesis) representation, improving dense retrieval quality by bridging the distribution mismatch between queries and documents.

C. RAG on Technical and Industrial Documentation

Applications of RAG to technical documentation include question answering over legal texts, enterprise knowledge bases, and specialized technical corpora [21]. For industrial robotics, prior work has focused on task planning from natural language [8]. Robot programming documentation presents

unique characteristics: high terminological specificity, version-dependent validity of information, and the requirement to produce structured, executable output rather than prose responses. Zhou et al. [26] demonstrate that retrieving documentation at query time (DocPrompting) significantly improves code generation for unseen libraries—directly applicable to PAC, where pretraining coverage is heterogeneous and unverifiable across models. Generating syntactically valid programs in proprietary industrial control languages from documentation-grounded RAG, and validating them against a real industrial IDE, is addressed only recently and in limited scope: Koziolek et al. [24] apply RAG with proprietary function block libraries for IEC 61131-3 Structured Text, and Fakih et al. [23] demonstrate that state-of-the-art LLMs fail to generate valid PLC code without compiler-guided verification pipelines. PAC presents an analogous challenge: even when an LLM produces compilable code, the absence of traceable documentation sources prevents engineers from verifying correctness for their specific hardware configuration.

D. Chunking Strategies for RAG

Chunking significantly affects downstream retrieval and generation quality. Qu et al. [13] evaluate semantic chunking versus fixed-size splitting, finding that the computational overhead of semantic approaches is not always justified by retrieval gains. Wu et al. [18] provide the most comprehensive controlled study, finding that cross-file context length is the dominant parameter and that structure-aware chunking does not consistently outperform sliding-window approaches for source code; AST-based structural chunking has also been explored [14]. Both studies evaluate chunking on general-purpose corpora; our ingestion pipeline applies LLM-driven semantic validation and artifact correction specific to industrial PDF technical manuals, where extraction artifacts (fragmented table rows, broken PAC code examples) degrade index quality more severely than in text corpora.

E. Vector Search and Retrieval Foundations

Dense retrieval encodes queries and passages into continuous vector representations using dual-encoder frameworks, enabling semantic similarity search beyond lexical overlap [28]. The complementary sparse retrieval approach, BM25 [31], scores relevance via probabilistic term weighting and serves as a strong lexical baseline. In the RC7 Assistant, dense retrieval via pgvector handles the semantic gap between natural language queries and PAC technical terminology, while hardware-configuration metadata filtering provides an orthogonal precision layer not achievable through embedding similarity alone.

III. Related Work

The most directly related work is Alfonseca and Martín Colino [1], who evaluate four LLMs on code generation for IBM APL2 and EES. Their key finding—that no LLM generates useful code in either language, and that cross-LLM error correction is equally ineffective—establishes the baseline against which the present work is measured. Joel et al. [2] survey LLM-based code generation for LRPLs and DSLs, motivating the choice of compilation success rate in WinCaps III as the primary metric.

On the RAG side, Self-RAG [7] introduces iterative retrieval with self-reflection, and FLARE [22] generates active retrieval triggers, optimizing retrieval frequency orthogonally to the HyDE-based semantic gap reduction pursued here. Vake et al. [4] propose Hypothetical Prompt Embeddings (HyPE), which shifts hypothetical content generation to the indexing phase. However, HyPE assumes user queries follow predictable patterns anticipatable at indexing time—an assumption that does not hold for PAC programming queries, which combine a particular robot configuration, I/O profile, and motion sequence that cannot be anticipated offline. Lei et al. [16] construct RAG pipeline variants over Stack Overflow posts including adaptive HyDE; the key distinction is that Stack Overflow covers mainstream languages with substantial training representation, whereas PAC has no such coverage, making in-context retrieval grounding far more consequential than for well-represented languages.

The closest family of work is LLM-based code generation for proprietary industrial control languages. Koziolek et al. [25] conducted the first systematic study of ChatGPT for PLC/DCS control logic, finding that GPT-4 generates syntactically correct code in many cases but lacks awareness of proprietary function block libraries. Koziolek et al. [24] extend this with a RAG-based method integrating proprietary function blocks. Fakih et al. [23] show that LLMs fail to produce valid PLC programs without compiler and verifier feedback loops. Fares and Herbold [29] study LLMs on ABB's RAPID language, finding that few-shot prompting handles simple modifications but fails for structural rewrites. These works establish the LRPL/DSL compilation failure pattern and motivate the RAG-plus-validation architecture of the RC7 Assistant. The hallucination problem underlying these failures is well documented: Ji et al. [30] note that grounding generation in retrieved verified content is the primary mitigation mechanism—precisely the role of RAG in the present system.

IV. Empirical Motivation: LLM Unreliability on PAC

A central premise of this work is that current LLMs are not a dependable option for programming proprietary industrial languages from pretraining alone. Rather than assume this, we establish it empirically with a structured comparative evaluation of four current LLMs queried without in-context grounding, validated by compilation in WinCaps III. This section reports that study; its findings motivate the system presented in the remainder of the paper.

A. Protocol

Four current LLMs are evaluated (Table II), each recorded by model and version. Models are queried in their free tiers, in clean sessions where no manual is provided in context. This isolates in-context grounding: pretraining coverage cannot be controlled and is itself the heterogeneous phenomenon

TABLE I
BASELINE SCOPE: EACH TASK MEASURES ONE DIMENSION OF THE CLAIM UNDER TEST.

Task	Lang.	Dimension measured
syntax reference	EN	knowledge / access
A1 — basic joint motion	EN	usable code
A2 — digital I/O	EN	usable code
B1 — inspection routine	ES	usable code
probe: source of each instruction	—	traceability, verifiability

TABLE II
LLMS UNDER TEST. ALL BASELINE RUNS WERE CONDUCTED BETWEEN JUNE 15 AND JULY 13, 2026. THE GROUNDED SYSTEM'S GENERATOR ACCESSES GEMINI 3.5 FLASH VIA API AS `gemini-3.5-flash`.

LLM	**Provider**	**Model / version**	**Tier**
ChatGPT	OpenAI	GPT-4o Mini	Free
Gemini	Google	Gemini 3.5 Flash	Free
Claude	Anthropic	Sonnet 4.6	Free
Grok	xAI	Grok 4.3 Fast	Free

under study, so the baseline measures generation without in-context grounding rather than without any grounding. Accessed through public free-tier interfaces—with vendor system prompts, decoding settings and silent updates outside our control—the baseline is an ecological evaluation of public LLM products, not a controlled comparison of model weights; each model string, version and test date is recorded for partial replication.

The instrument covers four tasks, chosen so that each dimension of the claim under test is measured by a prompt built for it (Table I). The three code-generation tasks were selected purposively to exercise three distinct dimensions of the system: controller-specific motion syntax (A1), industrial I/O manipulation (A2), and multi-step, configuration-aware routine generation in a second language (B1); they are not claimed to represent PAC as a whole, and the evaluation is accordingly designed as an initial proof-of-concept validation of the implemented artifact rather than a comprehensive benchmark of PAC code generation. The syntax-reference task asks the model for a reference for motion control and produces no program: it isolates whether the model knows PAC at all, independently of whether it can write compilable code. A1, A2 and B1 are code-generation tasks spanning the primary categories—basic joint motion, digital I/O, and a multi-step motion routine—and are validated by compilation. Language is a declared variable per set: Set A is administered in English, Set B in Spanish, and prompts are not translated. Tasks that do not bear on these dimensions—header generation and the higher-complexity integration tasks—are outside this baseline and are noted in Section XI.

After each code-generation prompt, in the same session, a traceability probe asks the model to list every instruction it used and cite, for each one, the exact official DENSO source where its syntax is documented—manual title, section and page—with an explicit instruction not to generalize. The probe is the discriminating element of the design: it separates a model knowing something from a model being able to show that what it says is correct, and it mirrors, for the baseline, the citation-validity question (RQ3) posed of the system in Section VIII. Each generated program is loaded into WinCaps III (PAC V3.3.00) and validated with the `check grammar` function. Because LLM output is stochastic, each prompt is run three times and the per-run outcome is reported, which allows repeated-run consistency, rather than only single-shot capability, to be observed.

B. Results

The syntax-reference task establishes the first dimension. Asked for a reference for motion control, no model refused and none returned generic non-PAC content: all four produced recognizable PAC—program declaration and termination, speed setting, motion statements, apostrophe comments—and correctly distinguished PAC on RC7 from PACScript on RC8 and later controllers. The models attribute their answers to public aggregators such as `studylib.net` and `Scribd`, and to DENSO's own manual portal, consistent with partial PAC documentation being present in pretraining data, though the training corpora themselves are not observable. What none of them supplies, when the probe asks for the manual, section and page of each instruction, is a citation that can be checked: the answers name a manual without a page, give a page that does not document the instruction, or direct the reader to obtain the manual and look it up. Knowledge without provenance is the condition this baseline is built to expose.

The remaining three areas test whether that knowledge yields usable code. We report the compilation outcome of every run under the WinCaps III `check grammar` criterion. The compilation sample comprises three tasks: A1 (basic joint motion) and A2 (digital I/O) from Set A, administered in English, and B1 (a HOME→P1→HOME inspection routine on a DENSO VP-6242) from Set B, administered in Spanish. Each task was run three times per model. Table III gives the per-run error count; a count of zero denotes a program that compiles.

Across the 35 runs with compilation data, **one compiled without errors (2.9%)**: Grok on A1, run 1. Every other run failed `check grammar`. The four models fail in distinct, systematic ways, verified against the DENSO manuals. ChatGPT fabricated the motion instruction in every A1/B1 run (`MOVEJ`, `JMOVE`, `MOVJ`, `MOVL`) and the I/O interface in A2 (`DI`/`DO`), all reported as *Undefined name*. Claude used the real `MOVE` instruction with an invalid interpolation type (`MOVE J`; PAC uses P/L/C) and invented an object-style I/O API (`IO.GetDI`, `SetDO`). Gemini and Grok used more of the valid forms (`MOVE P`, `IO[n]`, `SET`/`RESET`) but failed on joint-data literals, undefined taught positions (`P_HOME`), and program structure. Consequently, an engineer without deep PAC expertise cannot predict which model will succeed on a given task, nor which of a program's instructions are fabricated.

TABLE III

PAC COMPILATION RESULTS FOR FOUR LLMS QUERIED WITHOUT IN-CONTEXT GROUNDING, VALIDATED BY `CHECK GRAMMAR` IN WINCAPS III (PAC V3.3.00). EACH CELL IS THE NUMBER OF COMPILATION ERRORS FOR ONE RUN; **0 ERRORS MEANS THE PROGRAM COMPILES**. MODELS (FREE TIER, CLEAN SESSIONS, THREE RUNS PER TASK): CHATGPT (GPT-4O MINI), GEMINI (FLASH 3.5), CLAUDE (SONNET 4.6), GROK (GROK 4.3 FAST). TASKS FOLLOW THE SET A/SET B SCHEME: *A1* BASIC JOINT MOTION AND *A2* DIGITAL I/O (SET A, ENGLISH); *B1* HOME→P1→HOME INSPECTION ROUTINE ON A VP-6242 (SET B, SPANISH). ACROSS THE 35 RUNS WITH DATA, ONLY ONE COMPILED (GROK, A1 RUN 1).

Task	Run	ChatGPT	Gemini	Claude	Grok
Set A (English)					
A1	1	4	2	5	**0**✓
A1	2	6	2	5	4
A1	3	5	2	5	7
A2	1	12	4	9	3
A2	2	11	1	11	13
A2	3	15	13	n/a*	4
Set B (Spanish)					
B1	1	7	4	11	1
B1	2	31	9	7	3
B1	3	7	17	13	3
Mean errors		10.9	6.0	8.2	4.2
Compiles (of runs)		0/9	0/9	0/8	**1/9**

*Claude A2 run 3 was not captured (incomplete free-tier session), could not be re-run under the same free-tier model version, and is excluded from Claude's counts; it is neither a pass nor a fail. Overall: **1/35 runs compiled (2.9%)**.

Outcomes were also inconsistent *within* a model across runs of the same prompt (e.g., Grok yielded 0/4/7 errors on A1; Gemini yielded 4/1/13 on A2, using the correct `IO[n]` API in two runs and reverting to a nonexistent one in the third). A single run is therefore not evidence of (un)reliability, which is why each prompt is run multiple times. Finally, when asked to cite the exact manual, section, and page for each instruction, no model produced verifiable page-level citations.

C. Error Taxonomy

Of a nine-category compilation-error taxonomy derived from observed WinCaps III errors, eight categories appeared across the baseline (only catastrophic translator crashes were not observed): wrong interpolation method (`MOVE J`); undefined name (fabricated instructions and undefined positions/variables); format violation; wrong operator; program-structure error; unclosed or uncorresponded conditionals (`IF/END IF`, `WHILE/WEND`, `DO/LOOP`); undefined line number; and unknown character (Kana/kanji or accented characters flagged when a comment used `;` instead of `'`). Additional observed error types include invalid data types, variable/constant type mismatch, missing include files, and improper program names. Notably, some conditional-block errors are cascade effects secondary to a primary fault: when the I/O API is valid, the same conditionals parse cleanly; when it is invalid, they cascade. Distinguishing primary from cascade errors is therefore recommended when tallying the taxonomy.

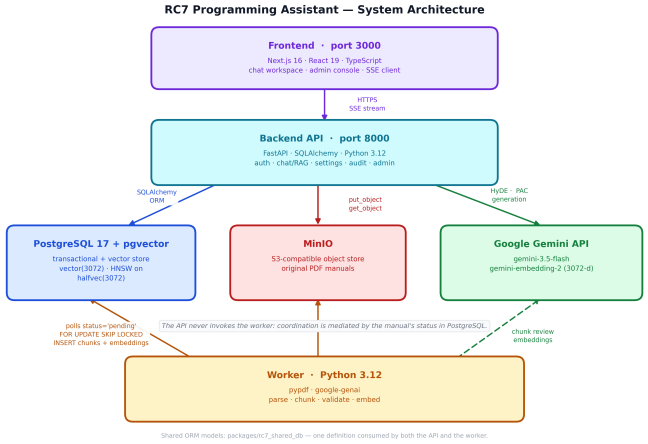

Fig. 1. RC7 Programming Assistant—system architecture. Dashed arrows: the worker calls the Gemini API for chunk validation and embedding generation.

D. Implication: Motivating Documentation Grounding

This baseline establishes the problem the RC7 Programming Assistant addresses. Queried without in-context grounding, current LLMs do not generate PAC consistently, compilably, and verifiably: only 1 of 35 runs compiled; each model fails in a distinct, systematic way; outcomes are inconsistent across runs of the same prompt; and no model produced verifiable documentation citations. In a setting where unverifiable robot code carries operational risk, ungrounded generation is not dependable as an RC7 programming tool. This motivates anchoring generation in retrieved official and installation-specific documentation—the DENSO RC7 manuals for instruction syntax, and workcell I/O tables, taught positions, and program banks that are never publicly available. Whether the complete documentation-grounded pipeline produces more consistently compilable and auditable PAC programs than ungrounded generation is the contribution evaluated in the remainder of this work.

V. SYSTEM ARCHITECTURE

The RC7 Programming Assistant follows a modular monolith architecture designed for single-instance laboratory deployment (Fig. 1). The system comprises three primary application components—a Next.js 16 frontend, a FastAPI backend, and a Python 3.12 worker—supported by PostgreSQL 17 with pgvector for unified transactional and vector storage, and MinIO for S3-compatible PDF object storage. All components are containerized via Docker Compose with declared health checks. Table IV summarizes the technology stack.

A. Design Decisions

The choice of pgvector over dedicated vector databases was motivated by operational simplicity at the target deployment scale. The DENSO RC7 manual corpus is bounded and

TABLE IV
SYSTEM COMPONENT SUMMARY.

Component	Technology	Responsibility
Frontend	Next.js 16 / React 19	UI, chat workspace, admin console, SSE client
Backend API	FastAPI / SQLAlchemy	Auth, RAG pipeline, settings, audit, admin
Worker	Python 3.12 / google-genai	PDF ingestion: parse, chunk, validate, embed
DB + vector	PostgreSQL 17 + pgvector	Transactional data + `vector(3072)` embeddings, HNSW index
Object store	MinIO (S3-compatible)	Original PDF manuals

relatively static; a dedicated vector store introduces overhead without measurable retrieval benefit at this scale [9]. Runtime-configurable parameters (generation and HyDE temperatures, request timeout, retrieval top-k, candidate pool, context budget, and the PAC system prompt) are persisted in a `system_settings` table and read at request time, enabling changes without service restarts. The ORM models are defined once in a shared package consumed by both the API and the worker, so schema definitions cannot diverge between services.

B. Observability and Audit

The system implements a fail-safe audit module recording authentication actions, administrative operations, manual lifecycle actions (upload, update, deletion), and chat query metadata. The audit trail is emitted by the API and therefore covers what users and administrators do, not what the worker does: ingestion progress is observable through the manual's status and the worker's logs, and is not itself audited. The audit service is designed to never raise exceptions: a logging failure cannot degrade core functionality.

VI. Document Ingestion Pipeline

The ingestion pipeline transforms raw DENSO RC7 PDF manuals into a queryable vector index via six stages: (1) *Upload*: the PDF is stored in MinIO and its metadata persisted in PostgreSQL, marked as pending ingestion; (2) *Parsing*: text extracted page-by-page using pypdf, and the PDF outline read to label each page with the manual section it belongs to; (3) *Structural chunking*: within each page, paragraphs are packed up to a character budget (1200 by default) and a paragraph exceeding it is sliced at fixed size; chunks never cross a page boundary and carry their section label as metadata; (4) *Semantic validation*: each chunk submitted to `gemini-3.5-flash` for coherence scoring and safe autofixes (merging a chunk with its successor, or rewriting extraction artifacts such as spurious line breaks and fragmented headers), gated by configurable thresholds and fail-safe to leaving the chunk untouched; (5) *Embedding*: validated chunks vectorized with `gemini-embedding-2` at 3072 dimensions, each prefixed with its section label so that the vector carries the context the cut discarded, batched at 100 texts per call; (6) *Indexing*: embeddings persisted in pgvector as `vector(3072)` and the manual marked `indexed` on success or `failed` otherwise. The pipeline targets text-layer PDFs and performs no OCR.

The API does not invoke the worker: coordination is mediated by the manual's status. The worker polls for pending manuals and claims one atomically (`SELECT ... FOR UPDATE SKIP LOCKED`), so concurrent workers never process the same manual, and jobs interrupted mid-ingestion are re-queued automatically at start-up.

The division of labour between stages 3 and 4 is deliberate, and it is where this pipeline departs from standard RAG ingestion. The cut itself is mechanical: no attempt is made to place boundaries by meaning. This is consistent with the evidence—Qu et al. [13] find the computational overhead of semantic chunking is not always repaid in retrieval gains, and Wu et al. [18] find structure-aware chunking does not consistently outperform sliding-window approaches. What is semantic is the *repair*: stage 4 interposes an LLM that reads each mechanically cut chunk and fixes what the cut broke, without modifying the source documents. The architecture is to cut mechanically and repair semantically, rather than to cut by meaning. It is particularly relevant for DENSO RC7 technical manuals, where PDF extraction frequently introduces fragmented table rows, merged columns, and broken PAC code examples.

VII. Four-Phase RAG Pipeline

The query-response pipeline is the central technical contribution (Fig. 2).

A. Phase 1 — Hypothetical Document Embedding (HyDE)

Upon receiving query q with robot configuration $c = (\textit{model}, \textit{controller version}, \textit{firmware})$, the system invokes Gemini to generate a hypothetical answer h without retrieved context. The hypothesis is generated at temperature 0 and truncated to its first 600 characters, so that a given query resolves to the same retrieval embedding, and therefore to the same sources across repeated runs: fixing the temperature stabilizes retrieval—and therefore provenance—though external embedding and inference services do not guarantee absolute determinism. The retrieval embedding is

$$\mathbf{e} = \mathrm{Embed}(q \parallel h). \tag{1}$$

This bridges the semantic gap between short technical queries and the longer manual excerpts that answer them—the exact gap identified by Alfonseca and Martín Colino [1] as a root cause of LLM failure in DSL/LRPL contexts.

The hypothesis only closes that gap if it imitates the right kind of document. HyDE's premise is that h lies closer to the target passages in embedding space than q does, but a technical corpus is not distributionally uniform: the DENSO manuals interleave reference entries (a statement, its syntax, a short example) with narrative introductions, typing tutorials, worked samples and practice exercises. A hypothesis written as prose is itself a piece of prose, and retrieves the narrative material—which describes programming without documenting

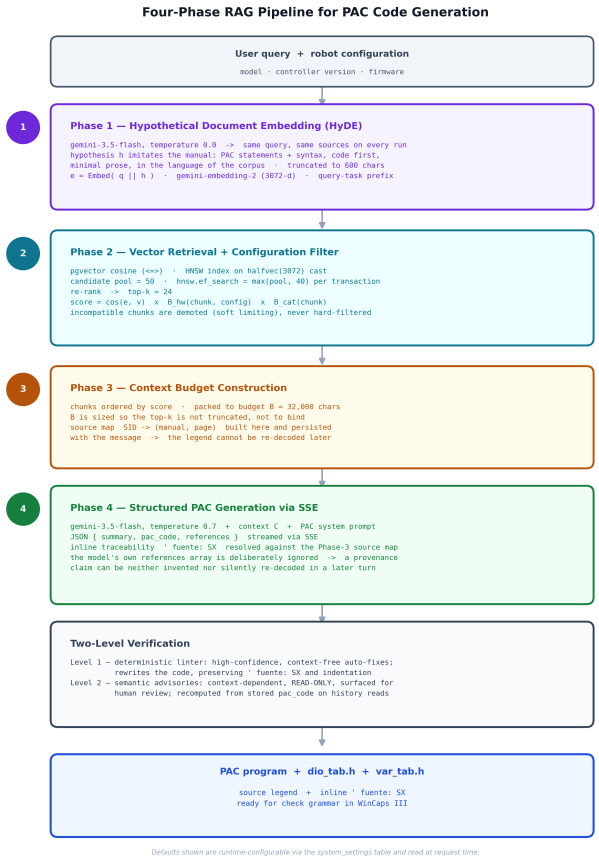

Fig. 2. Four-phase RAG pipeline for PAC code generation.

any statement. Phase 1 therefore instructs the model to impersonate the manual and emit the passage the manual would contain: PAC statements with their syntax and short inline comments, code first, minimal prose, and in the language of the corpus. Language matters independently of register: a hypothesis generated in the language of the query rather than of the manuals widens the very gap HyDE exists to close, and Set B queries are administered in Spanish against an English corpus. Both alignments—register and language—are properties of the Phase 1 instruction, not of the retriever, and cost nothing at query time.

B. Phase 2 — Vector Retrieval with Configuration Filtering

The embedding $\mathbf{e}$ queries pgvector using the cosine distance operator (<=>). Retrieval is served by a Hierarchical Navigable Small World (HNSW) index [27], a multi-layer proximity graph in which each vector links to its near neighbours: upper layers hold few nodes with long-range links, lower layers many nodes with short-range links, so a search descends from coarse to fine and reaches the neighbourhood of the query vector in a few hops instead of scanning every chunk. It is an approximate nearest-neighbour structure: it trades an exactness guarantee for sublinear search, and `hnsw.ef_search` governs that trade-off by setting how many candidates the graph explores. Because pgvector caps HNSW indexes on the `vector` type at 2000 dimensions, the index is built on a `halfvec(3072)` cast with `halfvec_cosine_ops`—`halfvec` stores each component in 16 rather than 32 bits, halving the index and bringing 3072-dimensional embeddings under the cap—and queries cast both sides to `halfvec` so the ordering uses the index rather than a full scan; `hnsw.ef_search` is raised per transaction to match the candidate pool. Postgres returns a wide candidate pool (default 50), which is then re-ranked to the top-k chunks (default $k = 24$) by a multiplicative score:

$$\text{score}(chunk_i) = \cos(\mathbf{e}, \mathbf{v}_i) \cdot \beta_{\text{hw}}(chunk_i, c) \cdot \beta_{\text{cat}}(chunk_i), \tag{2}$$

where β_{hw} compares each dimension of the manual's hardware metadata against the user's robot configuration c: matching the robot model or the controller boosts the chunk ($\times 1.30$ and $\times 1.15$ respectively), a present-but-different value mildly penalises it ($\times 0.70$ and $\times 0.85$), and absent metadata is neutral at 1.0 so unlabelled chunks are never degraded. As β_{hw} is the product of both dimensions, it ranges from 0.595 to 1.495. The term β_{cat} is a secondary nudge by document category, taking the highest boost among the manual's categories (programming 1.30, startup 1.15, errors 1.10, robot specifications 1.05) and 1.0 otherwise. Incompatible chunks are *demoted* rather than hard-filtered—a soft limiting strategy that prevents responses valid for one RC7 variant from being surfaced for incompatible hardware, while ensuring the context is never left empty when no manual matches the exact configuration.

C. Phase 3 — Context Budget Construction

Retrieved chunks are packed into a configurable character budget B (default $B = 32{,}000$ characters), ordered by $\text{score}(chunk_i)$, ensuring the assembled context fits within the effective context window while maximizing information density. The budget is sized so that the top-k chunks are not truncated in the worst case rather than to act as a binding constraint.

D. Phase 4 — Structured PAC Generation with SSE

The assembled context and original query are submitted to `gemini-3.5-flash` at temperature 0.7—generation, unlike retrieval, is not required to be reproducible—together with the resident rule set: a system prompt carrying PAC syntax rules, I/O macro conventions, and the required output format. The rule set is the second grounding channel and the counterpart of retrieval. Both are documentation, and they divide the work by what varies: the rules encode what holds for every PAC program on this controller family—that `MOVE` takes an interpolation method of P, L or C; that an output is asserted with `SET`, never by assignment—and are distilled once from the manuals rather than looked up per query. Retrieval supplies what the rules cannot: the statement this particular task needs, and the I/O tables, taught positions and program banks of this particular installation. Neither channel subsumes the other, and the system is not proposed as retrieval alone. The model returns JSON containing: (1) `summary`, a natural-language explanation; (2) `pac_code`, an executable PAC program including `dio_tab.h` and `var_tab.h`; and

(3) `references`, the source identifiers it claims to have used. The system prompt instructs the model to attach an inline source comment (`' fuente: SX`) to each instruction it draws from a retrieved passage; the apostrophe syntax is a valid PAC comment, so the annotation never affects compilation. Which instructions receive one is the model's decision and is not enforced—the pipeline governs what a reference resolves to, not where the model chooses to place it.

Crucially, the model's own `references` array is *deliberately ignored*. The authoritative mapping SID $\rightarrow$ $(manual, page)$ is the source map built in Phase 3 from the retrieved chunks, persisted with the message. The source legend prepended to the program is therefore constructed deterministically from retrieval, never asked of the model, so a provenance claim can be neither invented nor silently re-decoded in a later turn. This is what makes traceability a property of the pipeline rather than of model self-report.

E. Two-Level Verification

Generated code passes through two independent verification levels. *Level 1* is a deterministic linter that applies high-confidence, context-free auto-fixes and rewrites the code, reporting the number of fixes applied; inline `' fuente: SX` comments and indentation are preserved. At the time of this study the linter comprised four line-anchored rewrite rules—rewriting `IO[]=ON/OFF` statements to `SET/RESET`, rewriting `MOVE J` to `MOVE P`, and removing a nonexistent `J()` constructor—and fix counts are logged at generation time but not persisted with the message. *Level 2* produces semantic advisories that are context-dependent and therefore *read-only*: they are surfaced to the user for human review and never modify the program (for example, flagging an output actuation that immediately follows a pass-motion designator, where the robot does not stop precisely at the target). Advisories are recomputed deterministically from the stored `pac_code` when history is read, so they remain consistent across sessions. Both levels are rule lists designed to be extended as new failure modes are observed.

The response is streamed to the frontend via Server-Sent Events (SSE), with keepalive comments every 15 seconds to prevent proxy timeouts. Fig. 3 traces a complete query example through all four phases.

VIII. Evaluation

A. Research Questions

Section IV established that ungrounded LLMs are not a dependable option for PAC. This study therefore examines whether the complete documentation-grounded pipeline produces more consistently compilable and auditable PAC programs than ungrounded generation. Consistency alone is insufficient: a model may occasionally produce compilable code without grounding, but in safety-critical deployment the output must also be verifiable against authoritative sources [23], [30]. The evaluation of the RC7 Programming Assistant addresses three research questions:

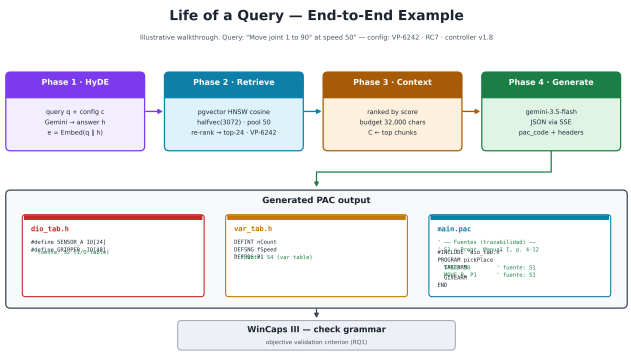

Fig. 3. Life of a query: end-to-end example traced through all four pipeline phases to compiled PAC output in WinCaps III.

- **RQ1 (compilation):** What proportion of PAC programs generated by the system compile without errors under the WinCaps III `check grammar` criterion, compared with the ungrounded baseline of Section IV?
- **RQ2 (error mitigation):** Which compilation-error categories observed in the ungrounded baseline remain in the system's final post-lint programs?
- **RQ3 (citation validity):** What fraction of the system's emitted inline source references resolves to a manual page that documents the cited instruction?

These questions rely mainly on objective compiler outcomes and structured manual checks, not on expert scoring of robot behaviour: compilation is deterministic (RQ1), while classifying an error (RQ2) and judging whether a cited page documents its instruction (RQ3) are structured checks against the manual that require only limited technical interpretation. This scopes the study to two of the three properties we distinguish—*traceable*, *verified* and *correct*; functional correctness lies outside it, and Section IX-C states what that costs. RQ3 carries the study's distinctive auditability claim; the metric reported is the precision of emitted citations, not the coverage of all generated instructions. The evaluation methodology follows the blueprint of Simon et al. [19] (baseline selection, qualitative failure analysis, transparent reporting) and the empirical grounding of Hasan et al. [20] for domain-specific RAG systems.

B. Procedure

The system is evaluated on the same three tasks administered to the ungrounded baseline in Section IV—A1 (basic joint motion, English), A2 (digital I/O, English) and B1 (a HOME→P1→HOME inspection routine on a DENSO VP-6242, Spanish)—using the prompt text verbatim, so that grounded and ungrounded generation are compared on equivalent tasks under an identical criterion. The robot configuration is supplied through the chat API rather than in the prompt body; this asymmetry between a configured system and a chat interface is unavoidable and is recorded in Section IX-C.

Each request to the system is independent: generation is conditioned only on the current query and the retrieved

context, and the persisted conversation history is never fed back to the model. The clean sessions that had to be enforced manually in the baseline are therefore a property of the system under test rather than of the protocol.

For each query, the generated `pac_code` is extracted, loaded into WinCaps III (PAC V3.3.00) and validated with `check grammar`; the outcome and any error messages are recorded verbatim. No expert scoring of functional behaviour is performed, in either arm. For RQ3, every inline source reference (`' fuente: SX`) is resolved through the persisted source map to a (manual, page) pair, and the cited page is then opened in the original PDF and inspected to determine whether it contains the instruction the reference is attached to. A citation counts as verified when its resolved page contains the cited instruction (source validity and instruction coverage); the check does not assess whether the page entails the specific parameters used. As in the baseline, each query is run three times so that repeated-run consistency, rather than only single-shot capability, can be observed. All nine runs were executed against a single frozen system version: no setting, system prompt or corpus was modified during the round. To support replication, the verbatim evaluation prompts, robot configuration, linter rules, and system source will be made publicly available.

C. Results

RQ1 (compilation). All nine final post-lint programs compiled without errors (Table V), against 1 of 35 runs for the ungrounded baseline (Table III); Table VI aggregates the comparison. Because retrieval settings and the corpus were held fixed, the three runs of a task are near-replicates rather than independent trials, so the contrast is reported descriptively at the task level rather than as a per-run significance test: the system resolved all three tasks in every run, whereas no ungrounded model compiled on more than one task—and across the 23 baseline runs of A2 and B1, none compiled. The contrast also holds with the generation model nominally fixed: the system's generator (`gemini-3.5-flash` via API; free-tier and API snapshots may differ) compiled 0 of 9 runs as an ungrounded baseline, so the move from 0/9 to 9/9 reflects the complete RC7 Programming Assistant pipeline rather than merely a change of model. It does not, however, isolate any single component: resident rules, retrieval and the linter act jointly, and separating their contributions is left to the ablation in Section XI.

The instruction that resolves each task was the same across the three runs of every cell: `DRIVEA (1, 45), (2, -30)` for A1, `MOVE L, P[pInspect], S=30` for B1, and `SET`/`RESET IO[]` guarded by `IF IO[] = ON` for A2. Retrieval settings and the corpus were held fixed (Section VII, Phase 1), and the retrieved context was in fact identical across runs of a prompt; generation, however, remains stochastic at temperature 0.7, and it is under that stochasticity that the programs converged. Inter-run variation was confined to inert detail—the default `@0` designator, the interchangeable `END IF`/`ENDIF` forms, and the value of a `DELAY` used for loop pacing; in A1 the three programs are identical line by line. This is the property the baseline lacked, where outcomes varied within a model across runs of the same prompt.

TABLE V
PAC COMPILATION AND CITATION-VALIDITY RESULTS FOR THE RC7 PROGRAMMING ASSISTANT. COMPILATION RESULTS CORRESPOND TO THE FINAL POST-LINT PROGRAMS, VALIDATED BY `CHECK GRAMMAR` IN WINCAPS III (PAC V3.3.00). THREE RUNS PER TASK AGAINST A SINGLE FROZEN SYSTEM VERSION. CITATIONS ARE VERIFIED BY OPENING THE CITED PAGE OF THE ORIGINAL MANUAL.

Task	Run	Errors	Compiles	Citations verified
Set A (English)				
A1	1	0	✓	3/3
A1	2	0	✓	3/3
A1	3	0	✓	3/3
A2	1	0	✓	4/4
A2	2	0	✓	6/6
A2	3	0	✓	5/5
Set B (Spanish)				
B1	1	0	✓	6/6
B1	2	0	✓	6/6
B1	3	0	✓	8/8
Total		**0**	**9/9**	**44/44**

RQ2 (error mitigation). None of the nine taxonomy categories derived in Section IV remained in the final post-lint programs; eight of the nine appeared in the baseline. One attribution caveat applies: the 9/9 outcome is the output of the complete pipeline including the Level-1 linter, whose four rules target baseline error categories (e.g., rewriting `MOVE J` and `IO[]=ON`). Per-run fix counts were logged but not persisted, so pre-lint compilation status cannot be reconstructed, and part of the mitigation may be attributable to deterministic correction rather than to generation alone. In particular the *wrong interpolation method* error—reported by WinCaps III as `Wrong interpolation method designated. Kw(J)` and the most frequent motion failure in the baseline—does not occur: the system emits `DRIVEA`, the statement the manual documents for absolute axis motion, rather than a fabricated `MOVE J`.

RQ3 (citation validity). All 44 inline citations emitted across the nine runs—the references the model attached, which it is not required to attach to every instruction—were resolved through the source map and checked by hand against the cited page of the original PDF: in every case the cited page contains the instruction the reference is attached to (Table V). No emitted identifier resolved to a nonexistent page, and none was attached to an instruction absent from its resolved page. By contrast, no baseline model produced verifiable page-level citations when probed. Citations resolved to reference sections of the corpus—*Motion Control* and *I/O Port* in the Programmer's Manual I, *DELAY* and *MOVE* in the Startup Handbook, among others—as well as to worked program samples.

The system is fully implemented and containerized, and was exercised end-to-end in a laboratory environment using the WinCaps III installation and licence of the Robotics Laboratory (RobLab). No generated program was executed on a robot: validation stops at `check grammar`.

TABLE VI
RC7 PROGRAMMING ASSISTANT VERSUS THE UNGROUNDED BASELINE ON IDENTICAL TASKS UNDER AN IDENTICAL COMPILATION CRITERION. BASELINE FIGURES AGGREGATE TABLE III (FOUR LLMS, THREE RUNS PER TASK).

	Ungrounded baseline		RC7 Assistant	
Task	compiles	mean err.	compiles	mean err.
A1	1/12	3.9	**3/3**	**0**
A2	0/11	8.7	**3/3**	**0**
B1	0/12	9.4	**3/3**	**0**
Total	1/35 (2.9%)	7.3	**9/9 (100%)**	**0**
Verified citations	none		44/44	

IX. DISCUSSION

A. Implications for Industrial LLM Deployment

The syntax-reference task showed that current LLMs know PAC, and showed why: they attribute their answers to public aggregators, consistent with how partial DENSO documentation could reach pretraining. That coverage is nonetheless heterogeneous, unverifiable, and silent on the configuration-specific context of a given installation. A technically informed user could close the gap by hand, pasting the relevant manual excerpts into a capable LLM, and this work does not claim otherwise. What it claims is that doing so does not scale and does not audit. It does not scale because the relevant excerpt must be located first, in a corpus of 2,982 pages across eleven manuals, for every query; because workcell I/O tables, taught positions and program bank contents are not in those manuals and not in any public corpus; and because the excerpt that is pasted today is not the excerpt that will be pasted tomorrow. It does not audit because a pasted prompt leaves no record: the engineer who reviews the resulting program six months later has the code and nothing else. The system answers both with the same mechanism—retrieval from a fixed, verified corpus, at a fixed retrieval temperature, with a source map persisted alongside the program—which is what turns a plausible answer into one that can be checked against a page.

B. Why HyDE Over HyPE for PAC Programming

Vake et al. [4] demonstrate that HyPE outperforms HyDE in general-domain benchmarks by eliminating runtime LLM latency. However, HyPE precomputes hypothetical questions at indexing time, requiring query patterns to be predictable offline. PAC programming queries are fundamentally context-specific: they combine a particular robot model, controller version, firmware, and I/O profile with a specific programming task, which cannot be anticipated during indexing. HyDE was therefore preferred because it supports query-specific hypothesis generation at runtime, which aligns with the configuration-dependent nature of PAC programming queries; this is a reasoned design choice, not an experimentally demonstrated superiority over direct retrieval, HyPE, or hybrid alternatives, and the SSE streaming design offsets the practical latency cost.

That choice comes with a condition that the general-domain literature does not surface, because general-domain corpora are comparatively uniform. Where a corpus mixes registers—and an industrial manual set does, interleaving reference entries with introductions, tutorials, worked samples and unsolved practice exercises—HyDE inherits whichever register its hypothesis is written in. The failure is quiet: retrieval returns passages that are topically apt, plausibly about the task, and documentary of nothing. Nor is it a chunking or re-ranking problem, so it is invisible to the parameters those layers expose; it is fixed in the instruction that generates the hypothesis. We report it because the mitigation is free and the diagnosis is not obvious: for retrieval over technical documentation, the hypothesis should imitate the target register and be written in the language of the corpus rather than of the user.

C. Limitations and Threats to Validity

What compilation does not establish. `check grammar` measures syntactic validity. A program may compile, cite verified sources, and still not perform the intended task; no generated program was executed on a robot, and runtime validation on physical hardware remains future work. A natural progression is a validation ladder: syntactic validity through compilation (established here); behavioural correctness in a simulator or digital twin; safe operation in a controlled sandbox cell; and only then execution on production hardware. Each rung answers a question the previous one cannot, and the present study establishes the first. Compilation and traceability are necessary but not sufficient evidence of usefulness: that is the price of instruments that need no expert judgement.

Generalization. Both arms cover three tasks and a single robot family (DENSO RC7/VP-6242); the motivating study additionally covers four LLMs. The architecture is not PAC-specific—nothing in HyDE, retrieval, re-ranking or the source map encodes the language—but that generality is a design property, not a demonstrated one: no second instantiation was built or tested, and the claim that it transfers is therefore untested. Transfer is also not free. The corpus and the resident rules are data and settings, but the two verification levels are PAC-specific rule lists in code, and the validation criterion itself is a vendor tool: another language means another compiler. Generalization to other models, to PACScript on RC8 and later controllers, or to other hardware configurations requires revalidation. The tasks exercise a narrow slice of the corpus: across the nine grounded runs, citations resolved to only two of the eleven indexed manuals, so the result speaks to the retrieval of motion and I/O documentation rather than to the corpus as a whole. The corpus itself is specific to the RC7 manuals available at the time of this study. The robot configuration reaches the system through structured fields rather than the prompt body, whereas the baseline could only receive it as prose; this asymmetry is inherent to comparing a configured system against a chat interface. And the opaque identity of free-tier models limits exact replicability of the baseline, mitigated by recording the model string, version and test date.

Pretraining contamination. The non-eliminable threat to the baseline is that current LLMs already contain part of

the publicly available DENSO documentation—the syntax-reference task confirms it. We therefore frame the baseline as generation without in-context grounding rather than without any grounding.

Dependencies. The system depends on external API availability for all pipeline phases, consistent with findings by Sorstkins [15] on HyDE latency in edge deployments.

X. Conclusion

This paper presented the RC7 Programming Assistant, a RAG-based system for generating PAC code for DENSO RC7 robots. Its design responds to a problem we established empirically rather than assumed: queried without in-context grounding, four current LLMs compiled PAC in only 1 of 35 runs, each failing in a distinct and systematic way, with outcomes inconsistent across runs of the same prompt and no verifiable documentation citations. Current LLMs are therefore not a dependable option for programming proprietary industrial languages from pretraining alone. The system answers this by putting documentation in the generation context in two forms: a resident rule set distilled from the manuals, and passages retrieved per query by a four-phase pipeline—HyDE-augmented retrieval, HNSW vector search with multiplicative configuration-aware re-ranking, context budget construction, and streaming structured PAC generation—over 2,982 pages of verified DENSO RC7 documentation plus installation-specific sources, resolving each source reference to a retrieved passage through a map the model cannot fabricate. On the same three tasks, under the same criterion, the system compiled in 9 of 9 runs against 1 of 35 for the ungrounded baseline, produced the same resolving instruction in all three runs of every task, and emitted 44 instruction-level citations of which 44 were verified by hand against the cited page of the original manual—where no baseline model produced a verifiable page-level citation at all. In the evaluated RC7 tasks, the complete RC7 Programming Assistant pipeline—resident rules, per-query retrieval, structured generation, and deterministic linting—was associated with consistently compilable PAC code that carries verifiable provenance, whereas ungrounded generation was not; attributing the effect to any single component awaits the ablation of Section XI. What the study does not claim is equally definite: compilation and traceability are necessary but not sufficient evidence of usefulness, and whether these programs do what was asked is a question for runtime validation on hardware, along the simulation-to-hardware ladder of Section IX-C. The system is fully implemented and containerized; no generated program has yet been executed on a robot.

XI. Future Work

Future work includes: (1) an ablation quantifying how the two grounding channels divide the work, which the present study does not measure—the resident rules and retrieval are both documentation, but how much each contributes, and where the boundary between them should fall, is an open design question; (2) assessment of functional correctness through runtime validation on physical DENSO RC7 hardware, extending the evaluation beyond the syntactic validity and traceability established by WinCaps III compilation; (3) extension of both arms to the tasks left outside this study—header generation and the higher-complexity integration tasks; (4) extension to additional DENSO robot families and other proprietary industrial languages; (5) a controlled test of the register and language alignment argued for in Section IX-B, measuring how often the hypothesis retrieves the reference section that documents the task under each variant, together with filtering the corpus by section register so that practice exercises, front matter and indexes do not compete with reference entries for the context budget; (6) investigation of fine-tuned embedding models for PAC technical vocabulary and of alternative customization strategies such as few-shot prompting and parameter-efficient fine-tuning [17]; and (7) an ablation study of chunking parameters following Wu et al. [18].

Author Contributions

Following the CRediT taxonomy: **S. Oviedo-Seas**: conceptualization, methodology, software, formal analysis, validation, writing—original draft. **I. Trejos-Zelaya**: supervision, writing—review & editing. **M. J. Rodríguez-Molina** and **S. Castro-Mata**: investigation, data curation, and validation—administration of the ungrounded-baseline protocol, `check grammar` compilation of the baseline programs in WinCaps III, and documentation of the per-run results.

Acknowledgments

The authors thank Universidad CENFOTEC's School of Software Engineering (ESOFT) for its institutional support of this research. The WinCaps III software licence and the DENSO RC7 installation used in this work were provided by the Robotics Laboratory (RobLab) at Universidad CENFOTEC.

Disclosure on AI use: The software system presented in this paper was developed with the assistance of Claude Code (Anthropic). The authors also utilized Large Language Models to assist with linguistic refinement and literature organization. All technical content, system design, architectural decisions, implementation oversight, analysis, and conclusions were developed entirely by the authors.

References

[1] M. Alfonseca and A. Martín Colino, "Using Large Language Models to Generate and Correct Code Written in Low-Resource and Domain-Specific Programming Languages," *Academia AI and Applications*, vol. 2, 2026, doi: 10.20935/AcadAI8228.

[2] S. Joel, J. J. Wu, and F. H. Fard, "A Survey on LLM-Based Code Generation for Low-Resource and Domain-Specific Programming Languages," *ACM TOSEM*, Oct. 2025, doi: 10.1145/3770084.

[3] X. Gu et al., "On the Effectiveness of Large Language Models in Domain-Specific Code Generation," *ACM TOSEM*, vol. 34, no. 3, Art. 78, Feb. 2025, doi: 10.1145/3697012.

[4] D. Vake, J. Vičič, and A. Tošić, "Bridging the Question–Answer Gap in Retrieval-Augmented Generation: Hypothetical Prompt Embeddings," *IEEE Access*, vol. 13, pp. 129952–129961, Jul. 2025, doi: 10.1109/ACCESS.2025.3589499.

[5] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *NeurIPS*, vol. 33, pp. 9459–9474, 2020.

[6] L. Gao, X. Ma, J. Lin, and J. Callan, "Precise Zero-Shot Dense Retrieval without Relevance Labels," in *Proc. ACL*, 2023, pp. 1762–1777, doi: 10.18653/v1/2023.acl-long.99.
[7] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection," in *Proc. ICLR*, 2024.
[8] J. Liang et al., "Code as Policies: Language Model Programs for Embodied Control," in *Proc. IEEE ICRA*, 2023, pp. 9493–9500, doi: 10.1109/ICRA48891.2023.10160591.
[9] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.
[10] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.
[11] B. Rozière et al., "Code Llama: Open Foundation Models for Code," arXiv preprint arXiv:2308.12950, 2023.
[12] Y. Gao et al., "Retrieval-Augmented Generation for Large Language Models: A Survey," arXiv preprint arXiv:2312.10997, 2023.
[13] R. Qu, R. Tu, and F. Bao, "Is Semantic Chunking Worth the Computational Cost?" arXiv preprint arXiv:2410.13070, 2024.
[14] Y. Zhang et al., "cAST: Enhancing Code Retrieval-Augmented Generation with Structural Chunking via Abstract Syntax Tree," in *Findings Assoc. Comput. Linguistics: EMNLP 2025*, Suzhou, China, 2025, pp. 8106–8116, doi: 10.18653/v1/2025.findings-emnlp.430.
[15] A. Sorstkins, "Assessing RAG and HyDE on 1B vs. 4B-Parameter Gemma LLMs for Personal Assistants Integretion [sic]," arXiv preprint arXiv:2506.21568, 2025.
[16] F. Lei, M. El Mezouar, S. Noei, and Y. Zou, "Never Come Up Empty: Adaptive HyDE Retrieval for Improving LLM Developer Support," arXiv preprint arXiv:2507.16754, 2025.
[17] L. Freire, F. A. Andaló, and N. S. Detlefsen, "Exploring Different Approaches to Customize Language Models for Domain-Specific Text-to-Code Generation," arXiv preprint arXiv:2603.16526, 2026.
[18] X. Wu, J. Gong, G. Jahangirova, and J. M. Zhang, "How Does Chunking Affect Retrieval-Augmented Code Completion? A Controlled Empirical Study," arXiv preprint arXiv:2605.04763, 2026.
[19] S. Simon, A. Mailach, J. Dorn, and N. Siegmund, "A Methodology for Evaluating RAG Systems: A Case Study On Configuration Dependency Validation," arXiv preprint arXiv:2410.08801, 2024.
[20] M. T. Hasan et al., "Engineering RAG Systems for Real-World Applications: Design, Development, and Evaluation," in *Proc. SEAA*, LNCS vol. 16082, pp. 143–158, Springer, 2026.
[21] S. Siriwardhana et al., "Improving the Domain Adaptation of Retrieval Augmented Generation (RAG) Models for Open Domain Question Answering," *TACL*, vol. 11, pp. 1–17, 2023, doi: 10.1162/tacl_a_00530.
[22] Z. Jiang et al., "Active Retrieval Augmented Generation," in *Proc. EMNLP*, 2023, pp. 7969–7992, doi: 10.18653/v1/2023.emnlp-main.495.
[23] M. Fakih et al., "LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems," in *Proc. ICSE-SEIP*, 2024, pp. 192–203, doi: 10.1145/3639477.3639743.
[24] H. Koziolek et al., "LLM-based and Retrieval-Augmented Control Code Generation," in *Proc. LLM4Code@ICSE*, 2024, pp. 22–29, doi: 10.1145/3643795.3648384.
[25] H. Koziolek, S. Gruener, and V. Ashiwal, "ChatGPT for PLC/DCS Control Logic Generation," in *Proc. IEEE ETFA*, 2023, pp. 1–8, doi: 10.1109/ETFA54631.2023.10275411.
[26] S. Zhou et al., "DocPrompting: Generating Code by Retrieving the Docs," in *Proc. ICLR*, 2023.
[27] Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, Apr. 2020, doi: 10.1109/TPAMI.2018.2889473.
[28] V. Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering," in *Proc. EMNLP*, 2020, pp. 6769–6781, doi: 10.18653/v1/2020.emnlp-main.550.
[29] S. Fares and S. Herbold, "Utilizing LLMs for Industrial Process Automation: A Case Study on Modifying RAPID Programs," in *Proc. IEEE/ACM 48th Int. Conf. Softw. Eng.: Softw. Eng. in Practice (ICSE-SEIP)*, 2026, pp. 248–258, doi: 10.1145/3786583.3786869.
[30] Z. Ji et al., "Survey of Hallucination in Natural Language Generation," *ACM Computing Surveys*, vol. 55, no. 12, Art. 248, Mar. 2023, doi: 10.1145/3571730.
[31] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Found. Trends Inf. Retr.*, vol. 3, no. 4, pp. 333–389, 2009, doi: 10.1561/1500000019.
[32] DENSO WAVE INCORPORATED, *DENSO Robot V*/H*/XYC*/XR* Series Startup Handbook*, Aichi, Japan, 2007, rev. 2011.
[33] DENSO WAVE INCORPORATED, *DENSO Robot Programmer's Manual I: Program Design and Commands*, Aichi, Japan, 2009, rev. 2011.