



LLM-Powered Workflow Optimization for Multidisciplinary Software Development: An Automotive Industry Case Study

Downloaded from: <https://research.chalmers.se>, 2026-08-08 13:37 UTC

Citation for the original published paper (version of record):

Wang, S., Yu, Y., Barr, E. et al (2026). LLM-Powered Workflow Optimization for Multidisciplinary Software Development: An Automotive Industry Case Study. Fse Companion 2026 Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering: 795-805. <http://dx.doi.org/10.1145/3803437.3805251>

N.B. When citing this work, cite the original published paper.

LLM-Powered Workflow Optimization for Multidisciplinary Software Development: An Automotive Industry Case Study

Shuai Wang
Chalmers University of Technology
Gothenburg, Sweden
shuaiwa@chalmers.se

Earl Barr
University College London
London, United Kingdom
e.barr@ucl.ac.uk

Yinan Yu
Chalmers University of Technology
Gothenburg, Sweden
yinan@chalmers.se

Dhasarathy Parthasarathy
Volvo Group
Gothenburg, Sweden
dhasarathy.parthasarathy@volvo.com

Abstract

Multidisciplinary Software Development (MSD) requires domain experts and developers to collaborate across incompatible formalisms and separate artifact sets. Even with AI coding assistants like GitHub Copilot, this process remains inefficient: individual coding tasks are semi-automated, but the workflow connecting domain knowledge to implementation is not. Developers and experts still lack a shared view, leading to repeated coordination, clarification rounds, and error-prone handoffs. We address this gap with a graph-based workflow optimization approach that replaces manual coordination with LLM-powered services, enabling incremental adoption without disrupting established practices. We evaluate our approach on spapi, a production in-vehicle API system at Volvo Group involving 192 endpoints, 420 properties, and 776 CAN signals across six functional domains. The automated workflow achieves 93.7% F1 score while reducing per-API development time from approximately 5 hours to under 7 minutes, saving an estimated 979 engineering hours. In production, the system received high satisfaction from both domain experts and developers, with all participants reporting full satisfaction with communication efficiency.

CCS Concepts

• **Software and its engineering** → **Software development process management**; • **Computing methodologies** → **Artificial intelligence**.

Keywords

Multidisciplinary Software Development, Automation, Workflow Optimization, Large Language Model

ACM Reference Format:

Shuai Wang, Yinan Yu, Earl Barr, and Dhasarathy Parthasarathy. 2026. LLM-Powered Workflow Optimization for Multidisciplinary Software Development: An Automotive Industry Case Study. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26)*, July 5–9, 2026, Montreal, QC, Canada. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3803437.3805251>



This work is licensed under a Creative Commons Attribution 4.0 International License. *FSE Companion '26, Montreal, QC, Canada*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2636-1/26/07
<https://doi.org/10.1145/3803437.3805251>

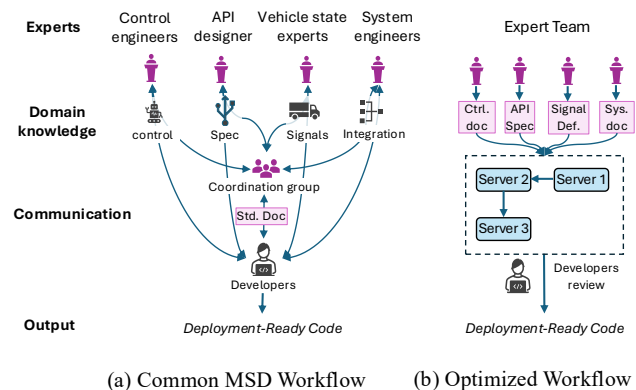


Figure 1: Comparison of workflows in the Multidisciplinary Software Development (MSD) process: (a) a typical MSD workflow and (b) optimized workflow through automated translation via our graph-based approach.

1 Introduction

In many industrial software projects, domain experts and software developers must collaborate across disciplinary boundaries, coordinating through heterogeneous artifacts that each party produces, maintains, and evolves independently. Experts contribute specifications, signal definitions, and design documents grounded in their respective domains, while developers must reconcile these artifacts and translate them into a unified, executable codebase. This coordination pattern is straightforward when artifacts are few and stable, but becomes a persistent bottleneck as systems grow in scope and the number of contributing disciplines increases. The automotive industry, with its deep regulatory requirements, safety-critical constraints, and multidisciplinary teams, offers an instructive example of how document-to-code translation can dominate engineering effort [31].

This pattern is characteristic of Multidisciplinary Software Development (MSD), which arises whenever software systems must encode specialized knowledge from non-software domains. In MSD settings, domain experts (engineers, scientists, regulators, or business analysts) produce specifications using discipline-specific formalisms that software developers must interpret, align, and implement as coherent system behavior [24]. The collaboration is

inherently two-way: developers translate domain requirements into code, while domain experts must understand implementation constraints to refine their specifications [15, 41]. This continuous dialogue creates value by embedding deep expertise into software, but it also introduces persistent operational challenges that scale poorly with system complexity.

This paper presents a practical case study in automotive software development at Volvo Group. We use *spapi* as a running example: an in-vehicle web server that exposes vehicle state and control-related properties through RESTful APIs to driver-facing mobile applications as well as backend services. Developing *spapi* requires a multidisciplinary team, including product owners, business translators, UI and app designers, Android engineers, software developers, control engineers working in MATLAB/Simulink, system engineers, and system architects. These roles rely on distinct technical vocabularies, toolchains, and artifact formats, and changes in one domain often propagate to others in ways that are difficult to anticipate or track. Using this case study, we highlight recurring operational problems in translating domain artifacts into code and show how LLM-based automation can substantially reduce manual overhead while maintaining delivery quality.

Figure 1 illustrates a common pattern we observe in MSD: requirements are authored across heterogeneous documents by different stakeholders, then manually translated by developers into implemented endpoints, followed by repeated feedback cycles where feasibility and constraints are communicated back to domain experts for refinement. In *spapi*, this translation spans a patchwork of documents and a correspondingly tedious patchwork of implementation tasks, which slows delivery and increases the risk of defects. Our goal, shown in Figure 1(b), is to automate the translation from domain artifacts to *spapi* endpoints. Achieving this requires transforming the MSD workflow itself: we reorganize the handoffs and dependencies so that LLM-powered services can replace repeated manual coordination. We design this workflow transformation using a graph-based representation of the artifacts and their dependencies. We make four contributions relevant to practitioners facing similar MSD challenges:

- We showcase the operational reality of artifact-driven multidisciplinary development at scale by characterizing the translation bottleneck in *spapi*, an in-vehicle API system at Volvo Group. The failure modes we identify generalize across industries where software must encode specialized domain knowledge, offering practitioners a vocabulary for diagnosing similar inefficiencies in their own workflows.
- We show that automating translation requires transforming the MSD workflow itself, not just accelerating individual coding tasks. We model the workflow as a graph of artifacts, dependencies, and handoffs, and use this representation to systematically restructure the process so that LLM-powered services can incrementally replace manual coordination while keeping domain experts in the loop and minimizing disruption to established practices.
- We provide quantitative evidence from a deployed system that generates 192 real-world API endpoints across six industrial domains. Compared to semi-automated implementations (human developers assisted by GitHub Copilot), the

automated workflow achieves a 93.7% F1 score and reduces per-API development time from approximately 5 hours to under 7 minutes, saving an estimated 979 engineering hours across the endpoint portfolio.

- We report deployment experience and stakeholder feedback from production use at a major automotive manufacturer. Domain experts and developers using the system daily reported high satisfaction (averaging 4.80 and 4.67, respectively, on a 5-point scale), indicating that the approach delivers practical value beyond offline metrics.

The rest of this paper is organized as follows. Section 2 describes the *spapi* system and its workflow challenges. Section 3 presents our optimization approach. Section 4 reports results and stakeholder feedback. Section 5 discusses practical considerations, Section 6 reviews related work, and Section 7 concludes.

2 The *spapi* System

spapi is an in-vehicle web server deployed at Volvo Group that provides an authenticated API layer mapping vehicle signals and control state to well-typed RESTful endpoints. Driver-facing mobile applications and backend fleet services consume these endpoints to display vehicle status, adjust comfort settings, and monitor operational parameters across the vehicle's functional domains.

Developing and maintaining *spapi* has required 15 to 20 full-time engineers (FTEs) to deliver more than 100 APIs across six functional domains: driver productivity, connected systems, energy management, vehicle systems, visibility, and dynamics. Three types of artifacts drive this workflow. OpenAPI specifications define endpoint behavior, data types, and interface contracts; they capture the essential complexity of the system and serve as the authoritative source for mobile and cloud integration [20]. Signal definitions describe low-level CAN bus messages that encode vehicle state, authored by control engineers and system architects with knowledge of the physical systems. Mapping documents bind high-level API properties to their underlying signals, specifying how abstractions like “climate mode” translate to specific CAN message fields.

The typical workflow proceeds through a sequence. Domain experts produce requirements and signal descriptions based on vehicle capabilities and business needs. Developers manually convert those documents into endpoints, adapters, data models, and tests, consulting the various artifact types to understand how each API property should behave. A dedicated coordination group, supplemented by direct contact with domain experts, resolves ambiguities that arise when specifications are incomplete or inconsistent. Under this workflow, defining a single API could take as long as 10 weeks from initial specification to deployment-ready code.

Observed Problems. Four problems characterize the workflow's failure modes, each contributing to inefficiency that compounds as the system scales.

Fragmented artifacts. Multiple overlapping documents, including API specifications, signal definitions, detailed descriptions, and coordination notes, describe related information in different formats and at different levels of abstraction. Developers must cross-reference and align these sources to produce correct implementations. When artifacts fall out of sync, as they frequently do during

requirements evolution, the reconciliation burden compounds and error opportunities multiply.

Manual-translation overload. Developers spend substantial effort transcribing information from specifications and signal definitions into code, drawing time away from higher-value work such as design, testing, and verification. Even with coding agents like GitHub Copilot, they must still locate relevant artifacts, interpret domain-specific formalisms, reconcile inconsistencies, and consult domain experts to resolve ambiguities. This burden scales with API and signal count. With over 400 unique properties and 776 associated CAN signals in the `spapi` system, translation remains a persistent bottleneck that coding assistants alone cannot remove; the workflow itself must be automated.

Ambiguity-driven churn. Underspecified domain documents trigger repeated clarification between developers and experts, consuming time and sometimes invalidating earlier implementation decisions. Signal definitions in `spapi` were often so concise as to be ambiguous, requiring consultation with vehicle state experts, system engineers, and sometimes control engineers, whose discipline has deep technical foundations spanning decades [5]. A single ambiguous signal could delay implementation by days.

Coordination bottlenecks. The coordination group created to streamline information flow in `spapi` development proved insufficient for managing hundreds of API properties and vehicle signals. Developers began bypassing it to contact domain experts directly, creating ad hoc communication patterns that were hard to track and prone to inconsistency. Once such workarounds start delivering value, organizational inertia makes them difficult to replace, even when their inefficiency is widely recognized.

Industry Implications. These problems are not unique to automotive systems. Similar bottlenecks arise wherever software must encode specialized domain knowledge and development requires ongoing coordination across disciplinary boundaries.

The consequences compound at scale. Manual transcription shifts engineering time away from architecture, testing, and verification, increasing both development costs and defect risk. Fragmented artifacts and frequent clarification cycles produce inconsistent implementations, with conflicts discovered late in integration when rework costs are highest. As product scope or regulatory requirements grow, staffing and coordination overhead increase disproportionately, yet adding engineers provides only temporary relief without addressing underlying inefficiencies. Perhaps most critically, domain understanding becomes concentrated in role-specific artifacts and individual expertise; when personnel change or specifications evolve, the implicit knowledge embedded in translation decisions can be lost, forcing costly rediscovery.

These failure modes motivate automation strategies that reduce manual handoffs through tighter artifact-code integration and targeted tooling. The following sections describe our approach and its application to `spapi`.

Our Approach. We address these challenges through an iterative workflow optimization approach that models the MSD process as a graph and progressively automates manual translation nodes. The workflow graph represents participants (domain experts, developers, and automated services) as nodes, with artifact exchanges as

edges. This representation makes explicit the communication structure that drives development effort and highlights where manual translation creates bottlenecks.

We apply a series of graph transformations that replace manual nodes with LLM-powered services, reducing complexity and communication overhead while preserving the information flows that domain experts rely upon. Each transformation targets a specific translation task: generating signal access code from definitions, aligning API properties with corresponding signals, and assembling complete endpoints from validated components. Rather than replacing the entire workflow at once, we incrementally automate individual tasks, validating each step with domain experts before proceeding to the next.

This graph-based perspective enables systematic identification of automation opportunities and provides a framework for measuring improvement across iterations. The result is a production pipeline comprising three coordinated services that reduce per-API development time from approximately 5 hours to under 7 minutes while achieving quality comparable to manual implementation.

3 Methods and Implementation

This section describes our approach to automating artifact-driven multidisciplinary workflows. We model the development process as a graph and apply iterative transformations that replace manual translation tasks with LLM-powered services.

3.1 Workflow Graph Representation

To formalize the complex interactions in software development, we represent the workflow as a directed dependency graph $\mathcal{G} = (\mathcal{V}, \mathcal{R})$. We define nodes $\mathcal{V}$ as concrete artifacts (e.g., specification documents, signal definitions, or constraint descriptions) and edges $\mathcal{R}$ as information dependencies.

Formally, let $\mathcal{V} = \{d_1, d_2, \dots, d_n\}$ denote the set of documents generated during the workflow. A relation $(d_i, r_{ij}, d_j) \in \mathcal{R}$ exists if the production or validation of document d_j depends on the information contained in d_i . The workflow $\mathcal{G}$ is thus represented as a set of relational triples:

$$\mathcal{G} = \{(d_i, r_{ij}, d_j) \mid d_i, d_j \in \mathcal{V}, r_{ij} \in \mathcal{R}\} \quad (1)$$

This formulation shifts the problem of workflow automation from managing human coordination to a structured transformation of a document dependency graph into executable code.

Figure 2 (Init) shows the initial workflow graph for `spapi`. Four expert roles participate: *API designers* responsible for OpenAPI specifications, *vehicle state experts* who define CAN signal semantics, *system engineers* who integrate signals across subsystems, and *control engineers* who establish vehicle behavior constraints. These experts produce artifacts that developers must translate into code, with a coordination group mediating communication. The graph is densely connected: all four expert roles interact with each other and with developers, resulting in significant coordination overhead. Without an explicit graph representation, such workflow transformations tend to remain ad-hoc, making coordination bottlenecks difficult to reason about systematically or evaluate across iterations.

This workflow graph $\mathcal{G}$ serves as the starting point for optimization. Our goal is to systematically replace manual translation

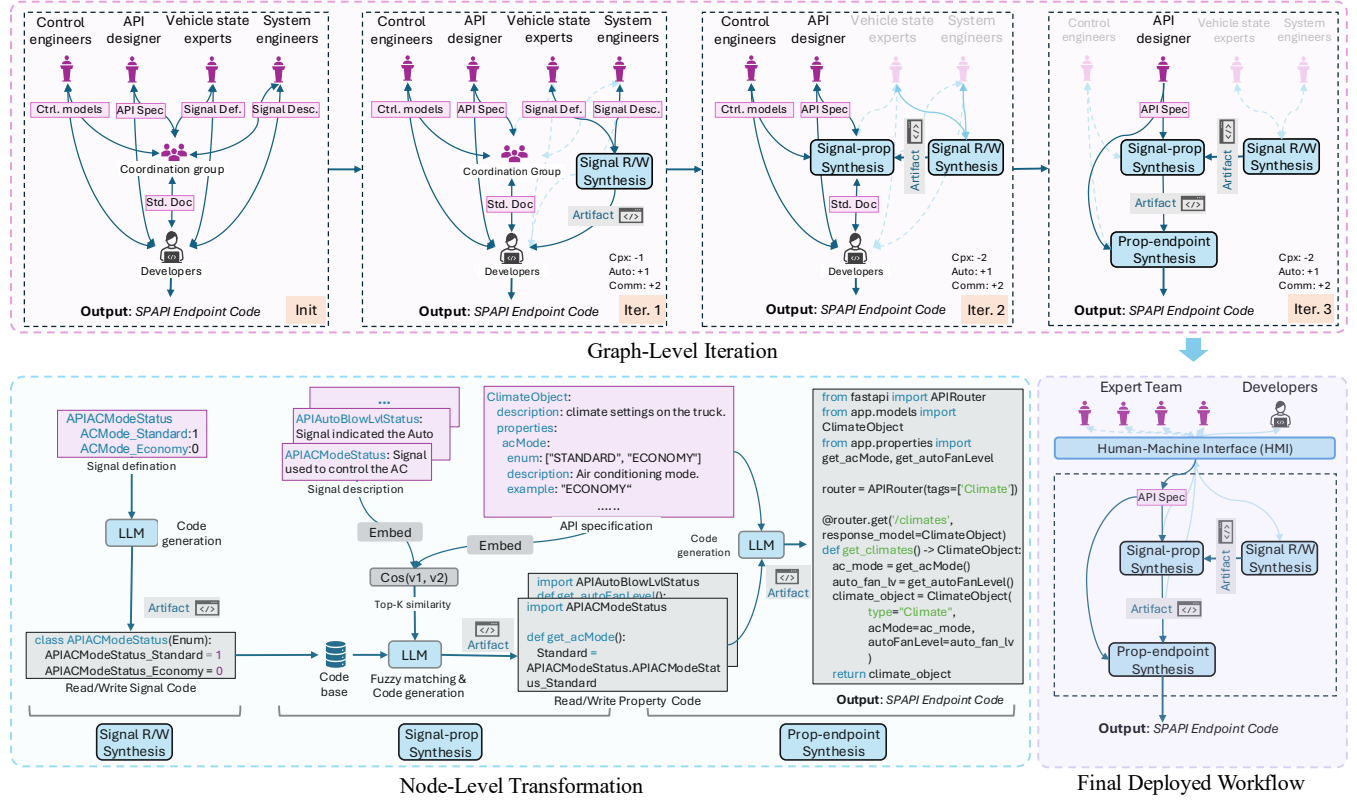


Figure 2: Iterative workflow optimization framework for vehicle API generation. The upper part illustrates graph-level optimization, depicting how the initial manual workflow undergoes three iterative refinements to achieve a highly automated workflow. The lower-right part illustrates the final deployed workflow, which comprises three server nodes. The lower-left part illustrates the detailed structure of each node.

nodes with automated services, reducing graph complexity and communication overhead while preserving the information flows that domain experts require.

3.2 Workflow Optimization Framework

Our framework aims to optimize the workflow by operating at two distinct levels: *node-level transformation*, which automates individual manual tasks, and *graph-level restructuring*, which streamlines the workflow by eliminating redundant coordination dependencies.

Node-level Transformation. At the node level, we formalize the conversion of manual translation tasks into LLM-driven automated services. For each document node $d_i \in \mathcal{V}$, we denote its associated input materials as $\mathcal{M}_i$ (e.g., domain specifications and signal definitions) and human-provided refinement instructions as $\mathcal{I}_i$. The transformation process is modeled as a function:

$$f_{\text{LLM}} : (\mathcal{M}_i, \mathcal{I}_i) \rightarrow c_i \quad (2)$$

where c_i represents a modular, executable code artifact. To ensure reliability, each c_i must pass an automated validation check Φ_{test} :

$$\Phi_{\text{test}}(c_i, \mathcal{M}_i) \rightarrow \{\text{pass}, \text{fail}\} \quad (3)$$

where Φ_{test} relies on test cases synthesized from $\mathcal{M}_i$. Only artifacts that satisfy $\Phi_{\text{test}} = \text{pass}$ are accepted. Upon successful validation, the manual node d_i is substituted by an automated service node s_i , which encapsulates c_i and exposes a programmatic interface. This substitution is denoted as $d_i \Rightarrow s_i$.

Graph-level Restructuring. Following node-level transformations, the initial graph $\mathcal{G}$ evolves into an intermediate state $\mathcal{G}'$. We then refine the topology by identifying and removing redundant coordination edges. An edge $(v_i, r_{ij}, v_j) \in \mathcal{G}'$ is defined as *redundant* if the information dependency r_{ij} can be resolved through direct programmatic invocation between their corresponding automated services.

Formally, we define an edge as *isRedundant* if: $\exists s_i, s_j$ such that $d_i \Rightarrow s_i$, $d_j \Rightarrow s_j$, and s_j can directly consume outputs of s_i .

The optimized workflow graph $\mathcal{G}^*$ is reached through an iterative reduction process:

$$\mathcal{G}^{(k+1)} = \{r \in \mathcal{G}^{(k)} \mid \neg \text{isRedundant}(r)\} \quad (4)$$

The process converges at a fixed point where $\mathcal{G}^{(k+1)} = \mathcal{G}^{(k)}$. Throughout this reduction, domain experts conduct reviews to ensure that no essential information flow is lost.

Table 1: Scoring scheme for transformation impact across three dimensions

Dimension	Score	Description
Complexity	-1	Task or exchange removed or combined
	0	No change
	+1	Task or exchange added
Automation	-1	Task becomes more manual
	0	No change
	+1	Task becomes more automatic
Communication	-2	Less explainable AND less executable
	-1	Less explainable OR less executable
	0	No change
	+1	More explainable OR more executable
	+2	More explainable AND more executable

3.3 Measuring Transformation Impact

To formalize the assessment, we model a transformation τ as a mapping from an original workflow W to a transformed workflow $W' = \tau(W)$. The overall impact of τ is evaluated along the three dimensions defined in Table 1.

Let the scoring function for each dimension $d \in \{\text{complexity}, \text{automation}, \text{communication}\}$ be $s_d(\tau) \in \mathbb{Z}$, where the range and semantics of s_d are given directly by the discrete levels in Table 1. Each score reflects expert judgment on how the transformation changes the corresponding property when comparing W and W' . The transformation impact vector is then defined as

$$\mathbf{s}(\tau) = (s_{\text{complexity}}(\tau), s_{\text{automation}}(\tau), s_{\text{communication}}(\tau)).$$

The first dimension measures graph complexity: whether a transformation adds or removes tasks and communication edges. The second measures automation level: whether work shifts from manual to automatic execution. The third measures communication quality through the lens of *duality*, i.e., the degree to which artifacts are both explainable to humans and executable by machines. Source code with meaningful identifiers and comments exemplifies high duality [10]. Transformations that produce executable code with clear documentation score positively on this dimension.

3.4 Node-Level Automation

We implemented three automated services to replace manual translation tasks in the spapi workflow. Each service corresponds to a distinct translation step, and together they form the pipeline shown in the lower portion of Figure 2.

3.4.1 Signal R/W Synthesis. The first service generates Python code for reading and writing CAN signals. It takes signal definitions as input (signal names, data types, value ranges, and enumeration mappings) and produces functions that interface with the vehicle's CAN bus.

```

1 class CANSignalFunctionWriter(dspy.Signature):
2     """Generate a Python function to read or write a CAN Signal.
3
4     - Read/write the CAN value or its enum equivalent.
5     - Enum class name = CAN Signal name (no changes).
6     - Function names: start with 'read_' or 'write_'.
7     - Check min/max if available.
8     - Include docstrings."""
9     CAN_Signal: dict = dspy.InputField(desc="CAN signal
10 metadata")
11     code: str = dspy.OutputField(desc="Generated Python module")

```

Figure 3: A simplified DSPy signature for Signal R/W Synthesis. The LLM generates Python functions to read or write CAN signals based on signal metadata.

Task: Align an API property with CAN signal(s). Let's think step by step:

- 1) Identify the API property intent (name/values/unit).
- 2) Select semantically equivalent CAN signal(s).
- 3) Check value domains; if mismatched, write an explicit mapping (e.g., ON/OFF $\leftrightarrow$ 0/1).
- 4) Check units; if different, annotate the required conversion (e.g., m/s $\leftrightarrow$ km/h).
- 5) If the property composes multiple signals, specify the one-to-many composition.

Return JSON as defined by: <DEFINED_JSON_SCHEMA>

Figure 4: Representative prompt used in the Signal-Property matching task (simplified for clarity).

We use the DSPy framework [21] for structured prompting, as illustrated in Figure 3. LLMs can be adapted to diverse tasks through prompting [7, 8] and excel at processing semi-structured inputs [19, 27]. DSPy's TypedPredictor ensures that generated code conforms to expected types; if output violates constraints, the system automatically re-prompts the model.

Recent work on constrained generation [6, 28, 35] enables LLM outputs to satisfy syntactic requirements, facilitating integration with traditional software tooling. For validation, we generate test cases using a separate LLM call and prompt the code-generating LLM to self-debug based on test failures. Once verified, signal R/W code is stored in a vector database indexed by structured signal descriptions, enabling retrieval for downstream synthesis steps. Our test-based validation approach aligns with recent work on LLM-driven test automation in automotive settings [37].

3.4.2 Signal-Property Synthesis. The second service aligns API properties with their corresponding CAN signal handlers. A key challenge in synthesizing deployable API endpoints is correctly aligning application-level API properties with underlying CAN signals. Given an API property, the system must identify the relevant CAN signals and determine how their values should be interpreted and composed to produce semantically correct behavior.

This is difficult because API and CAN specifications often diverge. Value domains may differ (e.g., ON/OFF vs. 0/1), units may not match (e.g., m/s on CAN vs. km/h in the API), and some properties aggregate multiple CAN signals (e.g., a time property derived from separate minute and second signals). So, naive matching by names or types is unreliable and may introduce silent semantic errors.

```

1 from fastapi import APIRouter, Depends, HTTPException
2 from pydantic import BaseModel
3 from typing import Any
4 router = APIRouter()
5
6 # ===== Models (Generated from OpenAPI) =====
7 class SpeedResponse(BaseModel):
8     speed: float
9     unit: str
10 # ===== Shared Dependencies =====
11 def get_auth_context() -> Any:
12     # Authentication / authorization logic
13     return {"role": "user"}
14 # ===== Endpoint Template =====
15 @router.get("/vehicle/speed",
16     response_model=Response)
17 async def get_vehicle_speed(
18     request: "SpeedRequest",
19     auth_ctx: Any = Depends(get_auth_context)):
20     try:
21         speed = read_vehicle_speed()
22         response = Response(speed=speed, unit="km/h",)
23         return response
24     except Exception:
25         raise HTTPException(status_code=500, detail="...")

```

Figure 5: Example of a FastAPI endpoint boilerplate template.

```

Task: Instantiate (not freely generate) a FastAPI
endpoint from an API specification.

Inputs:
- <FastAPI_Endpoint_Boilerplate_Template>
- <API_SPECIFICATION>

Constraints:
1) Treat the OpenAPI specification as the single
   source of truth.
2) Do not modify routes, methods or schemas.
3) Use the FastAPI template without modification.
4) Fill only endpoint-specific logic (property access
   and response composition).
5) Rely on FastAPI's schema-driven validation.

```

Figure 6: A representative prompt used in the API Endpoint synthesis task (simplified for clarity).

To make these assumptions explicit and verifiable, we use a JSON alignment template as an intermediate representation. The LLM must enumerate all contributing CAN signals and specify explicit value mappings, enum correspondences, and unit conversions when needed. As shown in Figure 4, generation is further guided by a chain-of-thought prompt encouraging stepwise reasoning over (i) property intent, (ii) value and unit consistency, and (iii) whether the mapping is direct, transformed, or composed from multiple signals. Together, the template constraint and structured reasoning reduce hallucinated assumptions and improve robustness across heterogeneous signal specifications.

3.4.3 Property-Endpoint Synthesis. The third service assembles property handlers into deployable API endpoints based on the OpenAPI specification. Given the API contract and generated property accessors, it produces complete FastAPI endpoint implementations, including routing, request validation, response serialization, and error handling.

Synthesizing endpoints directly from specifications is error-prone. Naively generated endpoints may violate the OpenAPI contract by using inconsistent parameter names, omitting or introducing response fields, or mismatching route paths and HTTP methods. Validation logic such as range checks or enum constraints may also be partially implemented or omitted, leading to subtle contract violations that are hard to detect at runtime.

To mitigate these risks, this service adopts a contract-first, template-based generation strategy. A FastAPI endpoint boilerplate template (Figure 5) fixes endpoint structure, routing, schemas, and cross-cutting concerns such as authentication, logging, and error handling. The LLM is restricted to instantiating endpoint-specific logic within predefined slots, guided by the prompt shown in Figure 6. Validation is delegated to FastAPI's schema-driven mechanisms, ensuring consistent enforcement of constraints specified in the OpenAPI document.

3.5 Graph-Level Optimization

Following node-level synthesis, the initial workflow is modeled as a coordination graph $\mathcal{G}^{(0)}$, where nodes represent development activities or automated services, and edges denote information dependencies and human-mediated coordination. Graph-level optimization is an iterative process that restructures $\mathcal{G}^{(k)}$ by introducing executable services to absorb coordination overhead. Formally, each optimization step applies a transformation $\tau_k : \mathcal{G}^{(k)} \rightarrow \mathcal{G}^{(k+1)}$ that replaces manual hand-offs with automated services, thereby eliminating edges resolved programmatically.

As illustrated in Figure 2, the transformation of the spapi workflow proceeds through three successive iterations:

Iteration 1: Signal R/W Synthesis. By generating executable signal access code directly from definitions, this service eliminates the requirement for developers to consult vehicle state experts, removing the associated coordination edge from $\mathcal{G}^{(0)}$.

Iteration 2: Signal-Property Synthesis. This stage automates the alignment between API properties and CAN signals. By generating mapping code, the service absorbs the complex interactions previously required between API designers, vehicle state experts, and system engineers, eliminating two additional edges.

Iteration 3: Property-Endpoint Synthesis. This transformation encodes vehicle-specific constraints (e.g., access permissions and value ranges) directly into OpenAPI annotations. The LLM utilizes these constraints during generation to ensure architectural compliance, removing the need for manual review by control engineers.

The cumulative impact of these transformations is quantified in Table 2. Each step τ_k consistently increments the automation level while enhancing communication quality by producing outputs that are both explainable and executable. Collectively, these optimizations reduced the workflow complexity by 5 edges, improved communication quality by 6 units, and increased the overall automation index by 3 relative to the manual baseline.

3.6 Deployed System

After iterative optimization, the final system replaces all manual translation steps with automated services and reaches a stable configuration validated by domain experts. The resulting workflow, shown in the lower-right of Figure 2, consists of three server nodes

Table 2: Cumulative transformation impact across three iterations.

Iteration	Comp.	Auto.	Comm.
1: Signal R/W	-1	+1	+2
2: Signal-Property	-2	+1	+2
3: Property-Endpoint	-2	+1	+2
Net change	-5	+3	+6

in a linear pipeline. Compared with the initial manual workflow, it achieves a net complexity reduction of 5, a communication quality improvement of 6, and an automation increase of 3.

System overview. The system takes as input an initial workflow graph $\mathcal{G}^{(0)}$ with dense manual coordination and a set of domain artifacts, including OpenAPI specifications, signal definitions, and constraint documents. It outputs an optimized workflow graph $\mathcal{G}^*$ in which manual translation nodes are replaced by service nodes and redundant coordination edges are removed, yielding a compact executable pipeline.

Architecture and interaction. Each node in $\mathcal{G}^*$ is deployed as an independent server with a well-defined API and composed sequentially to mirror the optimized workflow graph. The architecture decouples workflow logic from the underlying language model, enabling substitution of open-source or proprietary LLMs under different cost, latency, and privacy constraints. A unified human-machine interface allows domain experts to inspect API-to-signal mappings, review generated code, and request regeneration with added constraints, preserving human oversight without manual translation.

Execution and adaptability. The system supports both fully automated and human-in-the-loop execution. For well-specified tasks, the pipeline runs end-to-end without intervention; ambiguous mappings or specification inconsistencies are flagged for expert review. By replacing input artifacts or domain-specific instructions, the framework can be adapted to new APIs, vehicle platforms, or other translation-heavy workflows.

4 Evaluation

- RQ1: [Overall performance] Does the automated workflow achieve code quality comparable to AI-assisted human development (with Copilot) across different automotive domains?
- RQ2: [Reliability] Do the specific techniques employed (boilerplate templates, code composition, and automated debugging) each contribute measurably to system reliability?
- RQ3: [Efficiency] Does the automated workflow reduce development time compared to AI-assisted processes while maintaining acceptable quality?
- RQ4: [Human factors] Do domain experts and developers report satisfaction with the deployed system sufficient to support continued production use?

4.1 Experimental Setup

We conducted experiments on 192 real-world automotive API endpoints collected from six distinct industrial domains at Volvo Group,

as summarized in Table 3. These APIs collectively span 420 unique properties and 776 associated CAN signals, representing the full scope of the spapi system.

We use the original production implementations developed by engineers (assisted by GitHub Copilot) as ground truth, referred to as the *baseline*. Precision (P) measures the correctness of generated property-level code, computed as $P = \frac{|\text{Correct}|}{|\text{Generated}|}$. Recall (R) measures coverage of the original API specification, computed as $R = \frac{|\text{Correct}|}{|\text{Baseline}|}$. F1 is the harmonic mean of precision and recall, computed as $F1 = \frac{2PR}{P+R}$.

The baseline implementations were reviewed and deployed in production and thus reflect accepted system behavior. In this domain, semantic errors typically appear as enum mismatches, value range violations, or missing signal bindings, all targeted by our validation and automated debugging mechanisms. We therefore adopt a conservative matching strategy, flagging ambiguous cases for human review rather than generating potentially incorrect code.

To address RQ4, we invited all practitioners with direct experience of both workflows to provide structured feedback: four domain experts and two developers. This complete set of qualified evaluators rated the system on role-specific criteria using a 5-point scale. Experts assessed communication effort, functional coverage, and implementation accuracy; developers assessed communication efficiency, debugging effort, and code maintainability.

Baseline workflow. Because no existing automated solutions target API development in industrial automotive settings, we compare our system with APIs developed by human engineers assisted by proprietary AI coding agents. In this baseline, engineers were allowed to use GitHub Copilot. We assess both API quality and development time.

LLM usage and time cost. The system separates offline processing from deployment-time execution to manage LLM usage efficiently. All compute-intensive LLM operations occur offline and do not affect runtime performance. In our experiments, we use GPT-4o (2024-05-13).

During offline processing, the LLM scans and filters the CAN database and derives read/write equations for relevant signals. In a full run involving 192 APIs and 776 CAN signals, the system made about 15,000 LLM calls over 20.6 hours, averaging 396 seconds and 80 calls per API.

During deployment, the generated read/write equations are reused. The LLM is invoked only for lightweight tasks such as matching API properties to signals and synthesizing the final endpoint. On average, each API requires about 5 LLM calls, taking roughly 10 seconds. This overhead is low enough for practical engineering workflows.

Table 3 summarizes code quality results across all six automotive domains. The automated workflow performs strongly throughout: four domains (Driver productivity, Energy, Visibility, and Dynamics) achieve perfect precision of 100%, meaning all generated property-level code was correct. The other two domains (Connected systems and Vehicle system) still achieve precision above 94%.

Recall is slightly lower, ranging from 86.5% to 95.7% across domains. This reflects our conservative property-to-signal alignment strategy, which applies strict filtering to retain only high-confidence

Table 3: Performance and satisfaction scores across six automotive domains.

Domain	Domain Experts	Num	Performance			Satisfaction	
			P	R	F1	Expert	Developer
Driver productivity	HMI/UX, business translators, regulators	34	1.000	0.957	0.978	4.93	4.86
Connected systems	Business translators, dealers, sales staff	53	0.948	0.907	0.927	4.76	4.69
Energy	Control/mechanical/electrical engineers, emissions	18	1.000	0.913	0.955	4.86	4.71
Vehicle system	System architects, embedded software engineers	34	0.974	0.886	0.928	4.77	4.62
Visibility	Mechanical engineers, designers, HMI/UX	14	1.000	0.925	0.961	4.88	4.75
Dynamics	Control engineers	39	0.983	0.865	0.920	4.75	4.55
Total	-	192	0.976	0.902	0.937	4.80	4.67

Table 4: Ablation study: impact of individual techniques on code quality and satisfaction.

Configuration	Performance			Satisfaction	
	P	R	F1	Expert	Developer
Full automated workflow	0.976	0.902	0.937	4.80	4.67
without boilerplate templates	0.968	0.895	0.930	4.74	4.45
without code composition	0.956	0.883	0.918	4.69	4.43
without automated debugging	0.911	0.841	0.875	4.33	4.15

Table 5: Signal code accuracy before and after automated debugging, by signal type.

Stage	Signal Type	Enum	Bool	Numerical	Object	Total
	Count	634	90	33	19	776
Initial LLM output		96.50%	96.70%	90.90%	89.50%	93.40%
After automated debugging		100%	100%	100%	100%	100%

matches. Properties with ambiguous signal mappings are flagged for human review rather than used to generate potentially incorrect code. Even so, average recall exceeds 90%, indicating broad coverage of the original specifications.

The overall F1 score of 93.7% shows that the automated workflow produces code quality comparable to baseline development across diverse automotive domains. Performance also remains stable across different expert roles: the system handles APIs requiring input from control engineers (Dynamics, Energy) as effectively as those involving HMI/UX designers or business translators.

Stakeholder satisfaction scores (Section 4.4) provide further validation: both experts and developers report high satisfaction and note that the system supports transparent monitoring through well-defined interfaces.

Answer to RQ1: Yes. The optimized workflow achieves 93.7% F1 with consistent performance across six diverse domains, meeting production quality standards.

4.2 RQ2: Reliability of Code Generation Techniques

We conducted an ablation study to quantify the contribution of each technique to overall system reliability. Table 4 shows performance when individual components are removed.

Removing boilerplate templates reduces F1 from 93.7% to 93.0% and decreases developer satisfaction from 4.67 to 4.45. Without

Table 6: Property-to-signal matching performance by embedding strategy.

Embedding Input	P	R	F1
Raw signal code	0.763	0.277	0.406
Original descriptions	0.861	0.451	0.592
Rewritten descriptions	0.980	0.925	0.952

templates, generated code exhibits inconsistent structure and style, requiring additional manual cleanup. Removing code composition (assembling endpoints from reusable, validated fragments) reduces F1 to 91.8% and satisfaction scores to 4.69 and 4.43 for experts and developers respectively.

The most significant impact comes from removing automated debugging. Without test-based validation and self-correction, F1 drops to 87.5%, and satisfaction scores fall to 4.33 and 4.15. This represents a substantial degradation that would likely be unacceptable for production use.

Table 5 provides detailed analysis of the debugging component. We categorized the 776 CAN signals into five functional types and measured accuracy before and after the debugging phase. Initial LLM outputs achieve high accuracy for most categories (96.5% for Enum, 96.7% for Bool) but lower accuracy for more complex types (90.9% for Numerical, 89.5% for Object). Common errors include subtle syntax mismatches, such as using `LowSupplyPress(Enum_Value)` instead of the correct `LowSupplyPress[Enum_Value]`. After automated debugging, all signal types reach 100% accuracy, validating our two-stage strategy of generating an initial draft followed by test-based refinement.

Effect of Embedding Strategies on Property-Signal Matching. We also evaluated the impact of embedding strategy on property-to-signal matching, a critical step in the pipeline. Table 6 compares three approaches: embedding raw signal code, embedding original textual descriptions from documentation, and embedding rewritten descriptions enriched with clarified semantics.

All configurations maintain high precision due to strict similarity thresholds, but recall varies substantially. Raw signal code performs poorly (F1 = 0.406) because code lacks semantic context. Original textual descriptions improve performance (F1 = 0.592), but brevity and ambiguity limit accuracy. Rewritten descriptions,

Table 7: Detection performance for common specification errors.

Error Type	P	R	F1
Out-of-range value	0.979	1.000	0.989
Invalid enum value	0.989	1.000	0.994

Table 8: Quality and time comparison: automated vs. baseline development.

Configuration	Performance			Time	
	P	R	F1	Per API	System Total
Full automated workflow	0.976	0.902	0.937	396s	20.6h
without Signal R/W Synthesis	0.965	0.913	0.939	+1.5h	+288.0h
without Signal-Property Synthesis	0.992	0.953	0.972	+2.5h	+480.0h
without Property-Endpoint Synthesis	0.980	0.907	0.942	+1.1h	+211.2h
Baseline workflow (Copilot)	0.959	0.906	0.932	+5.1h	+979.2h

where LLMs expand and clarify signal semantics, achieve F1 of 0.952, demonstrating that investment in description quality yields substantial returns.

4.2.1 Robustness to Specification Errors. To assess robustness in real-world conditions, we evaluated the system’s ability to detect errors in user-provided specifications. We manually injected two common error types into YAML specification files: numerical values exceeding defined ranges and enum values not present in the allowed set.

As shown in Table 7, the system detects these errors with high precision, achieving F1 scores of 0.989 and 0.994 respectively. Beyond ensuring correctness under ideal inputs, the pipeline serves as a validation layer that provides early feedback to help users identify and correct specification errors before they propagate to generated code.

Answer to RQ2: Yes, each technique contributes measurably to system reliability. Automated debugging has the largest impact, improving F1 by 6.2 percentage points and enabling 100% accuracy on signal code after refinement. Boilerplate templates and code composition each contribute smaller but meaningful improvements. The embedding strategy for property-signal matching also significantly affects performance, with enriched descriptions improving F1 from 0.592 to 0.952.

4.3 RQ3: Development Time Comparison

We compared our automated workflow against baseline development by systematically replacing each automated service with human engineers and measuring both quality and time. Table 8 presents the results.

The fully automated workflow completes API generation in 396 seconds per endpoint (approximately 6.6 minutes), while the baseline workflow requires approximately 5.1 additional hours per endpoint, a reduction of over 97% in development time. Across the full portfolio of 192 APIs, automation saved approximately 979 hours of engineering effort compared to the baseline workflow.

Baseline development achieves higher recall in some configurations, revealing a quality-efficiency tradeoff. When humans perform Signal-Property Synthesis with the help from Github Copilot,

Table 9: Detailed satisfaction ratings from domain experts and developers.

Role	Evaluation Criterion	Score	Average
Expert (n=4)	Communication efficiency	5.00	4.80
	Accuracy of implementation	4.89	
	Coverage of functional requirements	4.51	
Developer (n=2)	Communication efficiency	5.00	4.67
	Debugging effort	4.37	
	Code style and maintainability	4.64	

F1 increases from 93.7% to 97.2% because engineers can resolve ambiguous mappings that the automated system conservatively flags for review. However, this 3.5 percentage point improvement requires 2.5 additional hours per API (480 hours total across the portfolio), a tradeoff that is difficult to justify at scale.

Each automated service contributes meaningful time savings. Signal R/W Synthesis saves 1.5 hours per API by automating the translation of signal definitions into access code. Signal-Property Synthesis saves 2.5 hours per API – the largest contribution – by automating the cognitively demanding task of mapping API properties to signals. Property-Endpoint Synthesis saves 1.1 hours per API by automating endpoint assembly from validated components.

Answer to RQ3: Yes. Per-API development time decreases from 5.1 hours to under 7 minutes (97% reduction), saving approximately 979 engineering hours across 192 APIs.

4.4 RQ4: Stakeholder Satisfaction

We invited four domain experts and two developers who work with the spapi system to evaluate the deployed workflow. Each participant rated the system on three criteria using a 5-point scale. Table 9 presents the detailed results.

Both groups report high overall satisfaction, with experts averaging 4.80 and developers averaging 4.67. Most notably, all six participants gave perfect scores of 5.0 for communication efficiency, indicating unanimous recognition that the automated workflow effectively reduces coordination overhead—the primary pain point identified in the baseline process.

Experts rated accuracy of implementation at 4.89, reflecting confidence in the correctness of generated code. The slightly lower score for coverage of functional requirements (4.51) aligns with our quantitative finding that recall is somewhat lower than precision; some edge cases require manual handling. Developers rated code style and maintainability at 4.64, indicating that generated code meets their quality standards, though debugging effort received a slightly lower score of 4.37, suggesting room for improvement in error diagnostics.

The system is deployed in production at Volvo Group and continues to serve as the primary mechanism for API development. Practitioners reported improved transparency and reproducibility compared to the baseline workflow, particularly due to the ability to trace API-to-signal mappings and consistently regenerate code as specifications evolve.

Answer to RQ4: Yes. Despite the small sample size, which represents the complete population of practitioners directly involved in both workflows, all participants reported satisfaction levels supporting continued production use. The unanimous improvement in

communication efficiency and sustained production deployment provide converging evidence that the approach effectively addresses the coordination bottleneck motivating this work.

4.5 Threats to Validity

Internal validity. The ground truth may itself contain errors. We mitigate this by using production code that has undergone review and testing.

External validity. Our evaluation focuses on a single API system at one automotive manufacturer. While the 192 endpoints span six functional domains involving diverse expert roles (control engineers, HMI designers, system architects, business translators), generalization to other organizations or industries requires further validation.

Construct validity. Property-level F1 may not capture all aspects of code quality. We supplement quantitative metrics with stakeholder satisfaction ratings to address this limitation.

5 Discussion

On choosing workflows to automate. Although Section 3 provides technical criteria for workflow automation, selecting suitable workflows ultimately still requires informed judgment. Beyond technical feasibility, successful adoption also depends on organizational readiness, stakeholder alignment, and openness to change, all of which are often difficult to evaluate in advance. In the case of *spapi*, the workflow was not only well-scoped but also supported by a team willing to experiment and iteratively refine the process, which proved critical to achieving tangible and sustained benefits.

On the use of LLMs in workflow automation. Our results show that LLMs can support multiple stages of workflow automation, including parsing, translation, generation, and evaluation. However, effective use of LLMs still requires experience and engineering judgment, particularly when deciding whether a task should be handled by an LLM alone or by a hybrid solution combining scripts and LLMs. The iterative nature of our approach allows LLM usage to evolve alongside the workflow itself, as teams gradually refine these design choices and improve the balance between automation, control, and reliability over time.

On multidisciplinary collaboration. Multidisciplinary software development often suffers from unclear ownership between developers and domain experts. By representing workflows explicitly and enabling LLM-driven translations into domain-accessible forms, our graph-based approach improves transparency and participation. This makes workflows more iterable and shareable, helping move teams closer to practical joint ownership.

From a software engineering perspective, our findings suggest that translation-heavy MSD workflows constitute a distinct class of coordination-intensive processes. In such workflows, development effort is dominated not by algorithmic complexity, but by repeated artifact translation, handoffs, and clarification cycles across roles. Our results indicate that effective automation in these settings lies primarily in restructuring artifact flows and coordination structures, rather than optimizing individual coding tasks. We believe this perspective can inform the design of future LLM-assisted tools that operate at the workflow level rather than the function or file level.

6 Related work

Improving MSD. Although software is increasingly being adopted across diverse fields, research on improving MSD remains relatively limited [36, 38]. Previous research includes a survey of non-software engineers [24] to understand their perspectives on effective collaboration with software engineers, as well as a study [14] that uses activity theory to interpret sources of friction in MSD. Other studies have investigated specific types of multidisciplinary collaboration, such as those involving data scientists [9, 23] and machine learning engineers [25, 26, 32], emphasizing the nature of these collaborations and the sociotechnical challenges that arise. Additionally, some research has addressed sector-specific challenges in MSD, such as in healthcare [39] and automotive domains [18, 29], identifying collaboration issues and their associated costs.

AI in MSD. As AI becomes more prevalent in software engineering, recent research has explored how AI can help address collaboration challenges in MSD. One interview study [30] used the concept of shared mental models to analyze communication gaps between AI developers and domain experts. Other studies [33, 40] explored the use of prompting as a means for rapid prototyping among participants with varied workflows. Research in Human-Computer Interaction (HCI) has also examined multi-participant interactions with AI systems [16], including the development of guidelines [3] and taxonomies [12] for human-AI collaboration. Additional works have focused on designing improved interactions [1, 17] to enhance the agency of domain experts.

LLMs for REST APIs. In connection with the case study, an expanding body of research leverages LLMs for various aspects of REST API engineering. This includes generating API logic from specifications [11], creating API documentation from source code [13], producing tests for APIs [22, 37], and enabling APIs as tools for LLM applications [2, 4, 34].

In contrast to prior work that improves isolated tasks, we target the workflow that connects heterogeneous domain artifacts to implemented APIs. We model this MSD workflow as a graph and transform it so LLM-powered services can incrementally replace manual translation and coordination while preserving domain-expert involvement, demonstrated in production on *spapi*.

7 Conclusion

In this paper, we show that LLM-powered workflow automation works in production MSD settings. By modeling workflows as graphs and selectively automating translation steps, we reduce coordination overhead while preserving human oversight. The *spapi* deployment at Volvo Group validates this approach at scale. The pattern we address is not automotive-specific; it applies wherever software encodes domain knowledge.

Acknowledgment

This work was partially funded by the Autonomous Systems and Software Program (WASP), supported by the Knut and Alice Wallenberg Foundation, and the Chalmers Artificial Intelligence Research Centre (CHAIR). The authors thank Erdem Halil and Shaphan Manipaul Sam Kirubahar for their valuable technical contributions to the research foundations.

References

- [1] Mateen Ahmed Abbasi, Petri Ihanola, Tommi Mikkonen, and Niko Mäkitalo. 2025. Towards Human-AI Synergy in Requirements Engineering: A Framework and Preliminary Study. *2025 Sixth International Conference on Intelligent Data Science Technologies and Applications (IDSTA) (2025)*. doi:10.1109/IDSTA66210.2025.11202850
- [2] Adam Alami and Neil A. Ernst. 2025. Human and Machine: How Software Engineers Perceive and Engage with AI-Assisted Code Reviews Compared to Their Peers. *arXiv preprint (2025)*. <https://arxiv.org/abs/2501.02092>
- [3] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N Bennett, Kori Inkpen, et al. 2019. Guidelines for human-AI interaction. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–13.
- [4] Jayachandu Bandlamudi, Ritwik Chaudhuri, Neelamadhav Gantayat, Kushal Mukherjee, Prerna Agarwal, Renuka Sindhgatta, and Sameep Mehta. 2025. A framework for testing and adapting rest apis as llm tools. *arXiv preprint arXiv:2504.15546 (2025)*.
- [5] Stuart Bennett. 1993. *A History of Control Engineering 1930-1955* (1st ed.). Peter Peregrinus, GBR.
- [6] Luca Beurer-Kellner, Marc Fischer, and Martin T. Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. <https://openreview.net/forum?id=pXaEYzrFae>
- [7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs.CL]* <https://arxiv.org/abs/2005.14165>
- [8] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *arXiv:2303.12712*
- [9] Gabriel Busquim, Allysson Alex Aratijo, Maria Julia Lima, and Marcos Kalinowski. 2024. Towards Effective Collaboration between Software Engineers and Data Scientists developing Machine Learning-Enabled Systems. In *Brazilian Symposium on Software Engineering (SBES)*. Curitiba, Brazil.
- [10] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints (*ICSE-NIER '20*). Association for Computing Machinery, New York, NY, USA, 25–28.
- [11] Saurabh Chauhan, Zeeshan Rasheed, Abdul Malik Sami, Zheyang Zhang, Jussi Rasku, Kai-Kristian Kemell, and Pekka Abrahamsson. 2025. Llm-generated microservice implementations from restful api definitions. *arXiv:2502.09766 (2025)*.
- [12] Dominik Dellermann, Adrian Calma, Nikolaus Lipusch, Thorsten Weber, Sascha Weigel, and Philipp Ebel. 2021. The future of human-AI collaboration: a taxonomy of design knowledge for hybrid intelligence systems. *arXiv preprint arXiv:2105.03354 (2021)*.
- [13] Sida Deng, Rubing Huang, Man Zhang, Chenhui Cui, Dave Towey, and Rongcun Wang. 2025. LRASGen: LLM-based RESTful API Specification Generation. *arXiv preprint arXiv:2504.16833 (2025)*.
- [14] Zixuan Feng, Thomas Zimmermann, Lorenzo Pisani, Christopher Gooley, Jeremiah Wander, and Anita Sarma. 2025. When Domains Collide: An Activity Theory Exploration of Cross-Disciplinary Collaboration. *arXiv preprint arXiv:2506.20063 (2025)*.
- [15] Benjamin Guerineau, Chen Zheng, Matthieu Bricogne, Alexandre Durupt, Louis Rivest, Harvey Rowson, and Benoit Eynard. 2017. Management of Heterogeneous Information for Integrated Design of Multidisciplinary Systems. *Procedia CIRP* 60 (2017), 320–325. doi:10.1016/j.procir.2017.02.020 Complex Systems Engineering and Development Proceedings of the 27th CIRP Design Conference Cranfield University, UK 10th – 12th May 2017.
- [16] Muhammad Hamza, Dominik Siemon, Muhammad Azeem Akbar, and Tahsinur Rahman. 2024. Human-AI Collaboration in Software Engineering: Lessons Learned from a Hands-On Workshop. In *Proceedings of the ACM/IEEE International Workshop on Software-intensive Business (IWSiB '24)*. doi:10.1145/3643690.3648236
- [17] Jeffrey Heer. 2019. Agency plus automation: Designing artificial intelligence into interactive systems. *Proceedings of the National Academy of Sciences* 116, 6 (2019), 1844–1850.
- [18] Hans-Martin Heyn, Khan Mohammad Habibullah, Eric Knauss, Jennifer Horkoff, Markus Borg, Alessia Knauss, and Polly Jing Li. 2023. Automotive perception software development: An empirical investigation into data, annotation, and ecosystem challenges. *arXiv preprint arXiv:2303.05947 (2023)*.
- [19] Gonzalo Jaimovitch-López, Cèsar Ferri, José Hernández-Orallo, Fernando Martínez-Plumed, and María José Ramírez-Quintana. 2023. Can language models automate data wrangling? *Machine Learning* 112, 6 (2023), 2053–2082.
- [20] Frederick P. Brooks Jr. 1987. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer* 20, 4 (1987), 10–19. doi:10.1109/MC.1987.1663532
- [21] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhaman, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714 (2023)*.
- [22] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging large language models to improve rest api testing. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 37–41.
- [23] Miryung Kim, Thomas Zimmermann, Robert DeLine, and Andrew Begel. 2016. Characterizing the Roles of Data Scientists in a Large Software Company. In *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*. Austin, TX, USA.
- [24] Paul Luo Li, Amy J Ko, and Andrew Begel. 2017. Cross-disciplinary perspectives on collaborations with software engineers. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2–8.
- [25] Hanjun Luo, Chiming Ni, Jiaheng Wen, Zhimu Huang, Yiran Wang, Bingduo Liao, Sylvia Chung, Yingbin Jin, Xinfeng Li, Wenyuan Xu, XiaoFeng Wang, and Hanan Salam. 2025. HAI-Eval: Measuring Human-AI Synergy in Collaborative Coding. *arXiv preprint (2025)*. <https://arxiv.org/abs/2512.04111>
- [26] Lucy Ellen Lwakatara, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. 2019. A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation. In *Agile Processes in Software Engineering and Extreme Programming*, Philippe Kruchten, Steven Fraser, and François Coallier (Eds.). 227–243.
- [27] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv:2205.09911 (2022)*.
- [28] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. *arXiv:2405.21047 [cs.AI]*
- [29] Joakim Pernstål, R. Feldt, T. Gorschek, and D. Florén. 2019. Communication Problems in Software Development – A Model and Its Industrial Application. *International Journal of Software Engineering and Knowledge Engineering* 29, 10 (2019), 1497–1538.
- [30] David Piorkowski, Soya Park, April Yi Wang, Dakuo Wang, Michael Muller, and Felix Portnoy. 2021. How ai developers overcome communication challenges in a multidisciplinary team: A case study. *Proceedings of the ACM on human-computer interaction* 5, CSCW1 (2021), 1–25.
- [31] Shaphan Manipaul Sam Kirubahar and Erdem Halil. 2025. Generating APIs through Library Learning using Large Language Models. (2025).
- [32] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).
- [33] Hari Subramonyam, Divy Thakkar, Andrew Ku, Juergen Dieber, and Anoop K Sinha. 2025. Prototyping with prompts: Emerging approaches and challenges in generative ai design for collaborative software teams. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–22.
- [34] Chunliang Tao, Xiaojing Fan, and Yahe Yang. 2024. Harnessing llms for api interactions: A framework for classification and synthetic data generation. In *2024 5th International Conference on Computers and Artificial Intelligence Technology (CAIT)*. IEEE, 628–634.
- [35] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagan-deep Singh. 2024. SynCode: LLM Generation with Grammar Augmentation. *arXiv:2403.01632 [cs.LG]* <https://arxiv.org/abs/2403.01632>
- [36] Shuai Wang and Yinan Yu. 2025. iQUEST: An Iterative Question-Guided Framework for Knowledge Base Question Answering. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 15616–15628. doi:10.18653/v1/2025.acl-long.760
- [37] Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Automating a Complete Software Test Process Using LLMs: An Automotive Case Study. *arXiv:2502.04008 [cs.SE]* <https://arxiv.org/abs/2502.04008>
- [38] Xixi Wang, Miguel Costa, Jordanka Kovaceva, Shuai Wang, and Francisco C Pereira. 2025. Plugging Schema Graph into Multi-Table QA: A Human-Guided Framework for Reducing LLM Reliance. *arXiv preprint arXiv:2506.04427 (2025)*.
- [39] Jens H. Weber-Jahnke, Morgan Price, and James Williams. 2013. Software engineering in health care: Is it really different? And how to gain impact. In *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–4. doi:10.1109/SEHC.2013.6602469
- [40] Zhixiong Zeng, Shuai Wang, Nan Xu, and Wenji Mao. 2021. Pan: Prototype-based adaptive network for robust cross-modal retrieval. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*. 1125–1134.
- [41] Chen Zheng, Matthieu Bricogne, Julien Le Duigou, Peter Hehenberger, and Benoit Eynard. 2018. Knowledge-based engineering for multidisciplinary systems: Integrated design based on interface model. *Concurrent Engineering* (2018).