

Generation and Verification: Human-Agent Collaboration for Safety-Critical Automotive Software Engineering

Elizaveta Zinovyeva¹, Aiham Taleb¹, Nikita Kozodoi¹, Daniel Schleicher¹,
Amir Mahdi Namazi², Martin Kraus², Levent Kent¹, Brian Jensen^{1*}

¹Amazon Web Services, Germany, ²AUMOVIO, Regensburg, Germany
{zinovyee, aitalleb, kozodoi, dschlei, levkent, jensbri}@amazon.com
{amir.mahdi.namazi, martin.2.kraus}@aumovio.com

Abstract

Research on coding agents mainly focuses on generation quality. Yet, safety-critical automotive software makes two additional demands. It prioritizes verification, since every change is reviewed, traced to a requirement, and checked against industry standards such as MISRA and AUTOSAR before it can be committed. It is also a narrow domain, written against layered architectures, embedded resource constraints, and organization-specific interface conventions that are scarce in the data general-purpose LLMs are trained on. We present a coding assistant for automotive C/C++ that addresses both demands. First, it exposes three tools to the user over the Model Context Protocol (`generate`, `edit`, and `verify`), allowing developers to run `verify` on any code on demand rather than relying on the agent to check its own output. Second, rather than using a general-purpose model, we customize the underlying LLM for the automotive domain through prompt engineering and fine-tuning. We construct a dataset from automotive production code of AUMOVIO, where each function is paired with the requirement texts and Unified Modeling Language (UML) behavior it must satisfy. In a two-week pilot on active production projects, developers report `verify` as the most-used tool and confirm using it on their own code and not only on generated output. Comparing model customization strategies, 75% prefer the faster prompt-engineered configuration. MISRA- and AUTOSAR-inspired pattern checkers indicate that prompt engineering achieves a lower number of violations on C, while fine-tuning shows better conformance with architecture patterns for C++.

1 Introduction

Research on AI coding assistants is organized largely around generation quality (Jiang et al., 2024), asking whether a model produces code that compiles, passes tests, and resembles a reference implementation (Paul et al., 2024; Zheng et al., 2025). Automotive software development is adopting these assistants as car makers seek productivity gains (Hepp et al., 2025). Yet, they fall short of safety-critical requirements such as AUTOSAR (AUTOSAR, 2017) and MISRA-C (Association, 2013), which establish strict rules for reliability in automotive systems. Recent studies show that popular LLMs struggle to produce MISRA-compliant code by default (Umer, 2025), while hallucinations pose tangible safety risks (Pavel et al., 2025). This places two additional demands on coding assistants for automotive software.

The first demand is verification. In safety-critical settings, developers cannot accept LLM output blindly, since every change is reviewed, traced to a requirement, and checked against coding standards before it can be committed. The practical question is therefore not only whether the model can write compliant code, but how the agent should collaborate with the developer so that compliance can be verified, audited, and trusted.

The second demand is domain fit. Automotive code must guarantee deterministic execution under real-time constraints, follow specific architectural patterns (e.g., AUTOSAR layered architecture) rarely encountered during pre-training, operate under tight memory and resource constraints in embedded systems, and respect interface conventions particular to the organization that owns the codebase. A general-purpose model fits this

*Work done while at Amazon Web Services.

setting poorly, since the conventions it must follow are underrepresented in its training data (Cassano et al., 2024; Gu et al., 2025; Liu et al., 2023).

This paper introduces an agentic assistant for automotive software, built on the Model Context Protocol (MCP) (Anthropic, 2024), that addresses both demands. We address the first demand by exposing verification as a separate tool. Rather than providing a single code generation mode, the system exposes three tools: `generate`, `edit`, and `verify`. The first two generate and revise function bodies from a signature, documentation, and context such as imported functions and UML diagrams. This task is central to automotive workflows, where interfaces are specified upfront and implementations must follow established patterns. The third tool lets developers run the same automotive rule checks the agent uses internally, on demand and independently of generation. Most coding agents treat checking as an internal step that the agent itself decides to invoke. We argue that in safety-critical settings the developer must be able to trigger evaluation directly, because a check the agent can skip is not a reliable check.

We address the second demand by customizing the model to the domain. We build a dataset from an AUMOVIO’s production automotive code in which each function is paired with the context needed to implement it compliantly, including requirement texts resolved from AUMOVIO’s requirements-management system and the UML behavior the function must realize. On this dataset we compare the two customization strategies most used for code models (Shin et al., 2025; Freire et al., 2026): prompt engineering (PE) with general-purpose API models and domain-specific fine-tuning (FT) (Zirak and Hemmati, 2024), both scored against custom rule sets inspired by established automotive standards. PE here involves supplying the assembled context and the rule sets in full at inference time, rather than tuning instruction wording, so the two strategies differ in when the domain information reaches the model and not in what it contains. Because those rule sets are pattern matchers, we treat their output as pattern conformance rather than compliance, and report how that changes what the numbers support.

The main contributions are the following. First, we present an MCP-based architecture in which verification is exposed as a tool the developer can invoke directly, on any code and at any time, and articulate the corresponding division of la-

bor between human and agent in safety-critical review. Second, we implement a context engineering pipeline that assembles requirement texts, UML behavior, and API context into training and inference examples, which lets us compare PE against FT on the same domain information, differing in whether it arrives during training or at inference. We analyze how the two methods perform in terms of architecture and coding pattern violations and generation latency. Comparing against a human baseline of shipped production code, we find that generated C code scores better than code written by the engineers, and we show that this reflects what the checkers reward rather than a genuine quality advantage. Finally, we report results from a two-week pilot with nine developers from AUMOVIO and show how the three tools are taken up in a real-world production workflow.

2 Related Work

Code generation and editing are part of the Machine Learning on Code or Big Code research field (Allamanis et al., 2018). Code generation for automotive software dates back to the late 1980s and 1990s (Hanselmann, 1998; Hanselmann et al., 1999), where early systems used rule-based transformations; neural approaches have become feasible only recently with the advent of LLMs. Early neural code generation models such as Codex (Chen et al., 2021) and open-source counterparts demonstrated strong accuracy in general programming (Li et al., 2022; Wang et al., 2021; Ahmad et al., 2021), but struggle with domain-specific code due to gaps in specialized knowledge. Gu et al. (2025) show that general-purpose LLMs underperform in domain-specific contexts because they make limited use of domain-specific libraries, and their DomCoder approach recovers quality by injecting API knowledge through prompting and fine-tuning. Another work fine-tunes code models on domain-specific corpora (Liu et al., 2023) and finds that developers expect assistants to understand specific frameworks and libraries, and their two-stage CodeGen4Libs outperforms non-domain-specific baselines.

Which of the two customization strategies to prefer is unsettled. Shin et al. (2025) compare prompt engineering against 17 fine-tuned models on code summarization, generation, and translation, and find that prompting a stronger general-purpose model does not consistently win, while Freire et al.

(2026) report that LoRA fine-tuning beats few-shot prompting and retrieval on domain-specific text-to-code. This motivates our comparison of the two strategies on the same assembled context.

For safety-critical domains, researchers integrate verification and domain constraints into generation. [Fakih et al. \(2024\)](#)’s LLM4PLC combines fine-tuning with an iterative verification loop, improving code generation success rates from 47% to 72% and expert-rated quality from 2.25/10 to 7.75/10, while [Jiang et al. \(2025\)](#)’s ROCODE integrates static analysis to reach a 99.1% compilation success rate. Recent work targets automotive compliance specifically: [Kirchner and Knoll \(2025\)](#) integrate LLMs into automotive development with test-driven development and iterative refinement; [Umer \(2025\)](#) finds that popular LLMs (ChatGPT, Gemini, DeepSeek, Copilot) produce no MISRA C++ compliant code by default; and [Pavel et al. \(2025\)](#) identify high rates of syntax violations and API conflicts in GPT-4 variants that pose safety risks. Together, these indicate that domain-specific code generation requires both LLM adaptation and domain-aware validation. Such validation is typically placed inside the generation loop, where the model critiques and repairs its own output, but gains from self-repair are modest and bottlenecked by the model’s ability to judge its own code ([Olausson et al., 2024](#)), and models struggle to self-correct without external feedback ([Huang et al., 2024](#); [Stechly et al., 2024](#)).

A further stream studies coding assistants from a developer-productivity perspective, reporting faster task completion with GitHub Copilot ([Peng et al., 2023](#)) and measuring suggestion acceptance and satisfaction across a 400-developer enterprise rollout ([Bakal et al., 2025](#)). [Heydari \(2025\)](#) argues that such evaluations still under-weigh how well a tool adapts to user expertise and supports collaboration. Productivity and adoption therefore depend not only on generation quality but on how the assistant supports oversight, which is amplified in safety-critical settings.

A recent line of work reframes review of generated code as a trust problem rather than a diffing problem. [Heander et al. \(2026\)](#) find, across a participatory design study with industry practitioners, that trust calibration is the central challenge when reviewing agent-generated multi-file changes, and argue that tools should surface risk and confidence signals at the granularity where reviewers allocate attention. [Mitropoulos et al. \(2026\)](#) supply the

security argument for keeping the reviewer independent of the generator: automated review judgments are systematically swayed by how a change is framed, and an iterative attack that refines its framing against a local copy of the review pipeline succeeds in every case they test, since the attacker may retry while the defender has one chance to detect. Both motivate our design: verification the developer controls, rather than verification the generating system elects to perform. Our work differs from these lines in treating verification as a developer-invocable tool on par with generation, grounding the comparison in production code, and a pilot on an active automotive project rather than synthetic benchmarks.

3 Methodology

Our coding assistant centers on an MCP-based framework compatible with modern AI coding assistants such as Cline, Claude Code and Codex, extending them with automotive-specific tools. We compare two ways of customizing the underlying model: prompt engineering (PE), which supplies domain knowledge at inference time, and fine-tuning (FT), which absorbs it into the parameters. We evaluate **Qwen3-32B** ([Yang et al., 2025](#)) in three variants, PE, FT, and FT on curated redundancy-removed data, and use **Claude Sonnet 4** and **Amazon Nova Pro** in PE mode as baselines.

3.1 Dataset Preparation

We build a context engineering pipeline to process an AUMOVIO’s production automotive C/C++ code into a structured dataset for fine-tuning and evaluation. The data comes from multiple actively maintained projects, so the examples reflect industry practices and compliance requirements. The pipeline parses C/C++ files to isolate individual functions, then augments each with the context a developer would consult when implementing it: header declarations and the signatures and documentation of imported functions for API context, import statements and namespaces, and the requirement and UML artifacts described next. It keeps only functions whose documentation is complete enough to specify behavior, validates them against our custom rule sets, and analyzes complexity for balanced stratification. The pipeline yields over 6,000 function implementations stratified by complexity, of which 822 are held out for evaluation.

The step that distinguishes this setting from gen-

eral code generation is requirements traceability. In safety-critical automotive practice, an implementation is not merely expected to work; it must be attributable to a specific requirement, and that link is what makes review and certification possible. AUMOVIO’s code carries this link in a machine-readable form: Doxygen comments cite requirement identifiers, while the requirement texts live in a separate requirements-management system. Our pipeline parses DOORS (IBM, 2025) export XML to expand each cited identifier into its full requirement text, and likewise extracts referenced PlantUML (PlantUML, 2025) diagrams to supply behavioral specifications. The pipeline also reconciles implementation signatures against their header declarations, correcting mismatches that would otherwise train the model on inconsistent interfaces. Each function’s prompt therefore states the complete obligation that the function is meant to satisfy.

Listing 1 shows the resulting format of the constructed function prompt, with the requirement text and UML fragment resolved in place. Appendix B details the extraction and filtering steps.

3.2 Model Customization

In PE mode, model parameters stay fixed and all domain knowledge enters through the prompt. Each request carries a system prompt that states the automotive coding role and instructs the model to return a single function implementation based on the task context assembled by our pipeline: the function signature, its documentation and linked requirements, signatures and documentation of imported functions, and the UML diagram when one is referenced. The prompt also includes JSON

rule sets the code needs to conform with, including conforming and non-conforming examples (see Section 3.3 for details). The rule sets are supplied in full rather than retrieved per request, since our subset is compact enough to fit the context window of all models we test, which keeps the prompt identical across examples and avoids confounding retrieval quality with model capability. Because the assembled context is the same one used to build training examples, the two strategies work from the same domain information, differing in whether it arrives at inference time or during training.

In FT mode, the same information is instead absorbed into the parameters. We select Qwen3-32B as the base model and train it on the preprocessed function corpus formatted as user-assistant turns via HuggingFace chat templates, with the context as the user message and the human production implementation as the target, so the model learns the conventions its authors applied rather than a synthetic notion of compliant code. We use Low-Rank Adaptation (Hu et al., 2021) with rank-8 adapters to adapt the model while preserving general coding capabilities, with 4-bit quantization (Dettmers et al., 2023) and gradient checkpointing to fit available memory, on SageMaker with DeepSpeed (Rasley et al., 2020) for multi-GPU training (Appendix C). We train two variants: one on the full corpus and one on a curated version with near-duplicate functions removed. The resulting models are packaged through Amazon Bedrock’s custom model import functionality for integration with the multi-model MCP architecture. At inference, the FT models receive the same assembled context and rule sets as the PE models, so FT adds parameter adaptation on top of the PE condition.

Listing 1: Abbreviated training example. Each numbered block is assembled by the pipeline from a different artifact in AUMOVIO’s repositories; requirement texts are resolved from DOORS identifiers cited in the Doxygen comment. Appendix D gives the full untruncated example.

```

1 INPUT CONTEXT
2 (1) Signature: void ValidateEngineParameters(
3     const EngineData_t* engineData, ValidationResult_t* result)
4 (2) Doxygen:    @brief Validates engine parameters ...
5               @requirements REQ_ENG_001, REQ_ENG_002 @safety_level ASIL-D
6 (3) Requirements (resolved from DOORS):
7     REQ_ENG_001: "... shall validate that engine RPM remains within the
8                   operational range of 800 to 6000 RPM ..."
9 (4) UML (PlantUML, if referenced):
10    ValidateEngineParameters -> CheckRPMRange : validate RPM
11 (5) API context: signatures + docs of imported functions, header declarations
12 (6) Rules (optional): JSON rule sets, see Listing 2
13
14 OUTPUT
15 if ((engineData == NULL) || (result == NULL)) { ... return; } /* null guard */
16 if ((engineData->rpm < 800U) || (engineData->rpm > 6000U)) /* REQ_ENG_001 */
17 { result->errorCode = VALIDATION_RPM_OUT_OF_RANGE; return; }

```

3.3 Automotive Rule Sets Integration

We build custom rule sets inspired by established automotive coding standards, covering patterns common in safety-critical embedded systems and prioritized by AUMOVIO’s engineers.

The **Coding** rules, 57 for C and 107 for C++, address pointer validation before dereferencing, buffer bounds checking, error return handling, and memory access patterns. The **Architecture** rules, 37 for C and 30 for C++, address layered architecture principles, interface design conventions, and module separation patterns typical in automotive software. Each rule carries one of four severity levels (info, warning, error, critical), mirroring the tiers used in static analysis tools. We review the rule sets with automotive engineers to confirm they target the patterns AUMOVIO prioritizes. The same rule sets serve two purposes: they supply context during code generation, and they act as scoring rubrics during evaluation (Section 4).

We process the rule sets into a JSON format (see Listing 2) holding rule identifiers, descriptions, and pairs of conforming and non-conforming examples.

3.4 Deployment

We package the solution as an MCP-based agentic coding assistant with three core tools, each mapping to a stage of the developer’s workflow: `generate` produces new function implementations aware of custom rule sets from a specification or signature; `edit` makes contextual modifications using existing code for refactoring or extending; `verify` checks code against the rule sets.

In the standard MCP arrangement, an agent reads tool descriptions and selects tools autonomously, so a well-described checker is invoked whenever the agent judges a check useful. We retain this setup but also expose `verify` to developers directly. The concern is not that agents skip checks; it is that the model which produced a piece of code is poorly placed to judge whether that code warrants

scrutiny. Safety-critical review practice resolves this through independence, where a different party controls the check. Exposing `verify` as a peer tool the developer invokes on demand preserves that independence.

Our architecture (Figure 1) integrates with existing development environments while maintaining separation between orchestration and specialized generation models. This separation is what makes the assistant agentic: the developer works in an IDE plugin such as Cline, whose orchestrator model decides which MCP tool a task requires and dispatches the request rather than answering from its own weights. Requests route through API Gateway, authenticated via Cognito, to Lambda functions that invoke the models we host for the automotive domain, so the orchestrator sequences the calls while specialized models generate code.

4 Evaluation Setup

Evaluating automotive code quality requires both automated metrics and human expertise. Table 2 in Appendix summarizes candidate metrics by type, tool availability, complexity, reliability, and cost, spanning reference-based similarity measures (Paul et al., 2024; Takaichi et al., 2022), static and dynamic analysis, and standards conformance against MISRA (Association, 2013), CERT C (Seacord, 2008), and AUTOSAR (AUTOSAR, 2017). We do not adopt the similarity measures as primary metrics, since closeness to a reference implementation is a weak proxy for the standards adherence developers must certify.

Our evaluation covers three dimensions. First, we score code against the rule sets of Section 3.3, comparing generated functions to a human baseline of production code written before the assistant was introduced on 822 held-out functions (574 C, 248 C++). Second, we measure generation latency. Third, we pilot the assistant with automotive engineers and survey their usage patterns.

Listing 2: Simplified C Example: Buffer Access Rule

```
1 {"rule_id": "BUF_001",
2  "rule_name": "Buffer Access Protection",
3  "non_compliant_example": "memcpy(dest, src, len);",
4  "compliant_example":
5  "if (dest & src & len > 0 & len <= DEST_SIZE & len <= src_size)
6  {
7  memcpy(dest, src, len); return SUCCESS;
8  }
9  return ERROR;"}

```

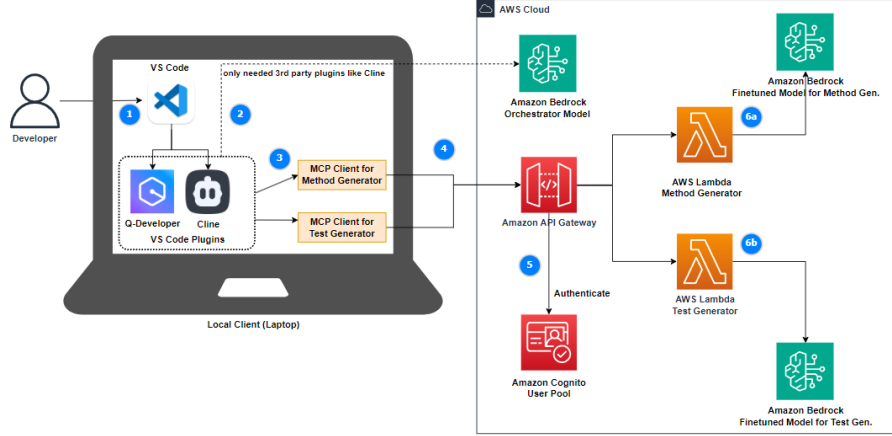


Figure 1: System architecture. The MCP server exposes three tools (`generate`, `edit`, `verify`) that AI coding assistants invoke. Requests route through API Gateway to models on Amazon Bedrock or SageMaker.

To score code automatically, we implement rule sets as pattern-based (regular-expression) static checkers. The rules they encode are drawn from verification patterns AUMOVIO already applies¹, selected by engineers as high-impact patterns. They are a deliberately simplified subset of AUMOVIO’s verification process: rules requiring semantic, data-flow, or inter-procedural reasoning are excluded by construction, since a pattern matcher cannot decide them and may produce false positives and false negatives. Production use therefore requires established automotive linters.

5 Results

5.1 Pattern Conformance Evaluation

Table 1 reports mean pattern-level rule violations per generated function alongside the observed end-to-end generation latency in seconds.

Holding the base model fixed at Qwen3 32B, PE gives the fewest C violations on both checkers (1.22 coding, 0.54 architecture), ahead of either FT variant. On C++ the ordering reverses, with FT on curated data recording the fewest coding (1.25) and architecture (0.02) violations. These margins are thin: the architecture column separates all three Qwen variants by at most 0.08 violations per function, which we do not treat as a meaningful difference. The same curated variant is also the worst Qwen variant on both C checkers (1.67 each), so its C++ gain comes with a C cost rather than a uniform improvement. Across model families, Claude Sonnet 4 (PE) and Amazon Nova Pro (PE) produce more violations in both languages than the

best Qwen3 32B variant.

One plausible reading of the C versus C++ divergence is that the two rule families differ in locality. Our C rules mostly describe function-local obligations, such as validating a pointer before dereferencing it or checking buffer bounds, which can be stated compactly in a prompt and applied by a general-purpose model without prior exposure to the codebase. The C++ architecture rules instead encode the AUMOVIO’s layering and interface conventions, which are longer-range, idiosyncratic to their projects, and largely absent from pretraining data. These are difficult to convey exhaustively in a prompt but appear repeatedly in training examples. If this reading is right, fine-tuning helps less by teaching C++ syntax than by absorbing organization-specific convention.

Observed generation latency ranges from ~ 15 seconds (Nova Pro, PE) to ~ 110 seconds (Qwen3 32B, PE). Note that these figures are not a controlled comparison of the customization strategies, since PE models are served through a managed API while FT models run on separately provisioned inference infrastructure. At the same time, the numbers characterize what developers in our pilot actually waited for when using the assistant. The spread is wide enough to motivate hybrid serving, with faster models reserved for interactive editing and slower ones for long-form function generation.

5.2 Comparison with the Human Baseline

Human production code records 2.45 and 2.20 violations per function on the two C checkers, worse than every generated variant. Read naively, LLM-generated C code adheres to automotive rules better

¹Licenses are required for the use of these standards.

Table 1: Automotive code generation: mean *pattern-level* rule violations per function (lower is better) and observed end-to-end generation latency. Counts reflect conformance to patterns our custom checkers can match, not MISRA/AUTOSAR compliance or functional correctness. Models are evaluated on 822 test examples (574 C, 248 C++) from production automotive code. **Bold** = best per column; underline = worst per column.

Model	Coding patterns (C)	Architecture patterns (C)	Coding patterns (C++)	Architecture patterns (C++)	Generation latency (s)
Qwen3 32B (PE)	1.22	0.54	1.73	0.10	<u>110.19</u>
Qwen3 32B (FT)	1.31	0.84	1.81	0.09	93.69
Qwen3 32B (FT, curated)	1.67	1.67	1.25	0.02	85.34
Sonnet 4 (PE)	1.80	1.84	2.61	0.12	28.68
Nova Pro (PE)	1.63	1.48	<u>2.88</u>	<u>0.24</u>	14.62
Human code (production)	<u>2.45</u>	<u>2.20</u>	0.84	0.06	N/A

than code written by the AUMOVIO’s own engineers and shipped in production vehicles. However, inspecting the outputs hints why this reading does not hold: generated functions apply defensive constructs, with null checks on pointer parameters, explicit bounds checks, single-exit control flow, and explicit error returns, and our checkers reward these surface patterns. Human code frequently omits a local null check because the invariant holds elsewhere, at a caller or through a module-level precondition that the author knows to hold and that a pattern matcher cannot see. The checker counts an absent local check as a violation whether or not the property holds non-locally, so it penalizes code that reasons beyond the function boundary and rewards code that re-checks everything. Such redundancy is not free in embedded settings, where extra branches cost code space and complicate timing analysis.

The C++ results point the other way, with human code best on coding patterns (0.84). Our C++ rule sets weight architectural and interface conventions more heavily than local defensive checks, so uniform guard insertion earns less credit there, while the AUMOVIO’s own code follows conventions its authors apply routinely. Together, the results suggest the checkers measure how closely code resembles a pattern vocabulary, which generated code matches well when the rules are local and less well when they encode organization-specific structure. The problem is that a pattern-matching proxy for a standard can be optimized without being satisfied: a model raises its score by emitting recognized constructs rather than by producing safer code. Such checkers are therefore limited but are still useful as a fast pre-commit filter and as generation context.

5.3 Pilot Study with Developers

Pattern conformance metrics describe what the models can produce in isolated tasks. To test the actual human-agent collaboration, we run a two-week pilot with nine of the AUMOVIO’s developers and analyze post-usage survey results.

Pilot participants are automotive engineers on AUMOVIO’s active C/C++ projects, recruited voluntarily from teams scheduled to trial AI assistance. Each installs the MCP server into Cline and works on their own production tasks rather than assigned benchmarks. All three tools are exposed through MCP descriptions, so the assistant can select them autonomously, and `verify` is additionally documented as directly invocable.

Evidence comes from a concluding survey covering experience, usage frequency, tool usage, usefulness, and other aspects. Every quantity is self-reported, since we collect no telemetry on invocation counts. Note that with nine respondents, we read only large gaps in the results as informative.

Developers rate their overall experience with the assistant 3.89 out of 5, and their willingness to recommend it to colleagues 4.00 out of 5. On average, the productivity improvements is marked at 4.67 on a seven-point scale. The distributions behind these means are more informative: no respondent gives the lowest rating on any item, but none is unanimous: experience draws 3 top ratings vs 1 low, recommendation – 4 top against 3 at or below the midpoint Figure 2 provides more detail on usability: 8 of 9 respondents find the assistant easy to integrate, and 5 of 9 rate its reasoning accurate against 4 neutral and none inaccurate. Usage does not settle at a middle value: four of nine respon-

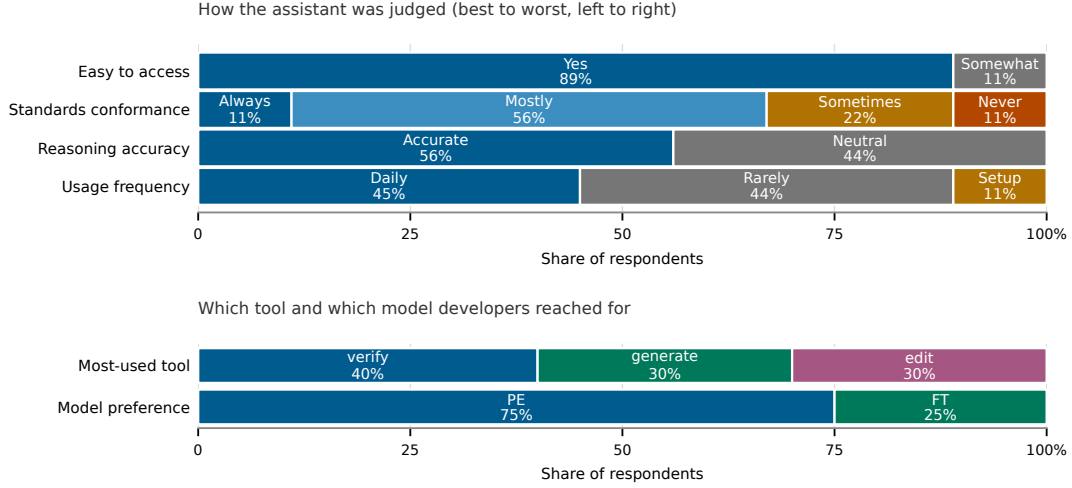


Figure 2: Pilot survey response distributions ($n=9$ developers). The upper panel holds judgments of the assistant, ordered from most to least favorable; the lower panel holds choices among tools and modes. Denominators differ by item, since not every respondent answered every question; raw counts are given in the text. Only large gaps are informative and percentages are rounded to sum to 100. “Setup” denotes participants who installed the assistant but reported no working use. All responses are self-reported.

dents use the assistant daily and four use it rarely, forming two distinct user populations.

Three patterns stand out in the survey results. First, 6 of the 8 respondents who answered this question prefer the PE mode. The PE configurations served in the pilot include the API-hosted models, which answer in roughly 15 to 29 seconds against 85 to 94 for the FT models (Table 1), so mode preference and response time are confounded here. Combined with developer feedback, we read this as evidence that in tight read-edit-review loops generation latency behaves as a key property.

Second, 6 of 9 developers report conformance *always* or *most of the time*, but only one of them answers *always*, while 2 answer *sometimes* and 1 *never*. This matters because in Table 1 the same models score at or better than human code on rule set conformance. Developers do not internalize the favorable automated numbers as trust.

Third, `verify` draws the largest share of most-used-tool responses (4 of 10 selections, against 3 each for `generate` and `edit`; the item permitted multiple selections) and the largest *very useful* share. Respondents also run `verify` on existing human-written code, not only on generated output, a use a generation-only assistant would not support.

6 Conclusion

AI assistance helps safety-critical development only if built around how such code is accepted: reviewed, traced to a requirement, checked before

commit. We present an MCP-based coding assistant addressing this by: (i) treating verification as a peer of generation rather than a step hidden inside it; (ii) adapting the model to the domain through prompt engineering and fine-tuning on production code paired with the requirements each function must satisfy. We deploy it on AUMOVIO’s active automotive C/C++ projects, evaluate pattern conformance, and run a developer pilot in production.

Our experiments produce three findings. First, `verify` is the most-used tool in the pilot, and developers run it on their own existing code rather than only on generated output, which indicates that verification is used as a first-class step. Second, with the base model held fixed, PE gives fewer C violations while the C++ counts separate the two strategies by margins too small to call, so we do not read either as the better choice. Third, developers prefer the faster configuration and do not fully trust the output despite its scores.

Together, these results suggest that coding assistants for safety-critical domains should be judged on more than generation quality. What mattered to our developers was checking code themselves, getting answers fast enough to stay in a review loop, and knowing where the automated verdict held.

Acknowledgments

We thank the wider AUMOVIO team who supported us throughout the project. We are especially grateful to Stefan-Cosmin Grigoriu for his support.

Limitations

Our evaluation has several limitations. Section 4 states what our pattern checkers can and cannot show, and Section 5.2 shows they favor generated code. We also report no functional correctness: no compilation or unit-test pass rates, and no check that a function satisfies the requirements or UML behavior in its prompt.

On the model comparison, we use parameter-efficient LoRA fine-tuning rather than full fine-tuning, and run no ablations over prompt design, context elements, or rule-set inclusion. We report no significance tests, so the small C++ margins should not be read as ranking the strategies. Latency was measured on different serving stacks, so it describes our deployment rather than the general strategies.

On the human evidence, the pilot has nine self-selected developers working with the assistant for two weeks. Everything is self-reported, with no telemetry and no record of whether generated code was committed or how much it was edited first, so the results are directional signals rather than effect estimates. Finally, all data comes from one organization, and the code, rule sets, and checkers are proprietary and cannot be released publicly.

Ethical Considerations

Defective code in the automotive setting can harm people, so the assistant is not meant to automate compliance work: everything it produces needs review by qualified engineers. Three failure modes matter in deployment. *False confidence*, where code that passes pattern checks still violates functional or timing requirements; Section 5.2 is a concrete case, since adding defensive patterns everywhere raises the score without making the code safer, and the extra branches can themselves affect code size and worst-case execution time. *Correlated blind spots*, where the generator and the checker share a model family and miss the same defect twice, which is why we keep the checker independent of the generator and why extending the architecture to test generation would need the same separation. *Conformance drift*, where non-conforming code spreads through reuse. Human approval gates, keeping the checker independent of the generator, and logging prompts, context, model IDs, and analyzer reports all help.

We ran no red-teaming and did not test whether the assistant can be pushed into producing insecure

code, tricked into evading the checkers, or made to introduce vulnerabilities our rules do not cover. Production readiness would require that work.

The study uses AUMOVIO’s proprietary code under their data governance. Pilot participation was voluntary, and we collected no personal data beyond anonymized survey responses.

References

- Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. [Unified pre-training for program and natural languages](#). *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2403–2417.
- Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.
- Anthropic. 2024. Model context protocol. <https://modelcontextprotocol.io/>. Accessed: 2025-01-15.
- Motor Industry Software Reliability Association. 2013. *MISRA-C: 2012: Guidelines for the Use of the C Language in Critical Systems*. MISRA.
- AUTOSAR. 2017. Guidelines for the use of the c++ 14 language in critical and safety-related systems.
- Gal Bakal, Ali Dasdan, Yaniv Katz, Michael Kaufman, and Guy Levin. 2025. Experience with GitHub Copilot for developer productivity at Zoominfo. *arXiv preprint arXiv:2501.13282*.
- Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. 2024. Knowledge transfer from high-resource to low-resource programming languages for code LLMs. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2).
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Tri Dao. 2023. [Flashattention-2: Faster attention with better parallelism and work partitioning](#). *Preprint*, arXiv:2307.08691.
- Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. [Flashattention: Fast and memory-efficient exact attention with io-awareness](#). *Preprint*, arXiv:2205.14135.
- Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

- Doxygen. 2025. Doxygen. <https://doxygen.nl/>. Accessed: 2025-08-12.
- Mohamad Fakhri, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros Araya, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. **Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems**. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP)*. ACM.
- Luís Freire, Fernanda A. Andaló, and Nicki Skafted Detlefsen. 2026. Exploring different approaches to customize language models for domain-specific text-to-code generation. *arXiv preprint arXiv:2603.16526*.
- Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. 2025. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22.
- Herbert Hanselmann. 1998. Development speed-up for electronic control systems. Technical report, SAE Technical Paper.
- Herbert Hanselmann, Ulrich Kiffmeier, Lutz Köster, and Martin Meyer. 1999. Automatic generation of production quality code for ecus. *SAE transactions*, pages 1625–1633.
- Lo Gullstrand Heander, Agnia Sergeyuk, Ilya Zakharov, Emma Söderberg, and Nikita Mukhortov. 2026. Trust-calibrated code review: A participatory design study of review workflows for LLM-generated multi-file changes. *arXiv preprint arXiv:2606.01969*.
- Dominik Hepp, Michael Amroudi, Martin Harrysson, Mateusz Wozniak, and Lukasz Maslaniec. 2025. From engines to algorithms: Gen ai in automotive software development.
- Soroush Heydari. 2025. A human centric requirements engineering framework for assessing GitHub Copilot output. *arXiv preprint arXiv:2508.03922*.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. **Lora: Low-rank adaptation of large language models**. *Preprint*, arXiv:2106.09685.
- Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*.
- Hugging Face. 2024. Chat templates. https://huggingface.co/docs/transformers/v4.47.1/en/chat_templating. Accessed: 2025-08-12.
- IBM. 2025. IBM engineering requirements management DOORS. <https://www.ibm.com/docs/en/engineering-lifecycle-management-suite/doors/9.7.0?topic=overview-doors>. Accessed: 2025-08-12.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.
- Xue Jiang, Yihong Dong, Yongding Tao, Huanyu Liu, Zhi Jin, and Ge Li. 2025. **Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation**. In *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. Research Track.
- Sven Kirchner and Alois C. Knoll. 2025. Generating automotive code: LLMs for software development and verification in safety-critical systems. *arXiv preprint arXiv:2506.04038*.
- Yujia Li, David Choi, et al. 2022. **Competition-level code generation with alphacode**. *Science*, 378(6624):1092–1097.
- Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. **Codegen4libs: A two-stage approach for library-oriented code generation**. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Kirchberg, Luxembourg. IEEE.
- Dimitris Mitropoulos, Nikolaos Alexopoulos, Georgios Alexopoulos, and Diomidis Spinellis. 2026. Measuring and exploiting contextual bias in LLM-assisted security code review. *arXiv preprint arXiv:2603.18740*.
- Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations (ICLR)*.
- Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2024. Benchmarks and metrics for evaluations of code generation: A critical review. In *2024 IEEE International Conference on Artificial Intelligence Testing (AITest)*, pages 87–94. IEEE.
- Marc Pavel, Nenad Petrovic, Lukasz Mazur, Vahid Zolfaghari, Fengjunjie Pan, and Alois Knoll. 2025. Hallucination in LLM-based code generation: An automotive case study. *arXiv preprint arXiv:2508.11257*.
- Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of AI on developer productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*.
- PlantUML. 2025. PlantUML language reference guide. <https://plantuml.com/guide>. Accessed: 2025-08-12.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. [Zero: Memory optimizations toward training trillion parameter models](#). *Preprint*, arXiv:1910.02054.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506.

Robert C Seacord. 2008. *The CERT C secure coding standard*. Pearson Education.

Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2025. Prompt engineering or fine-tuning: An empirical assessment of LLMs for code. In *Proceedings of the 22nd IEEE/ACM International Conference on Mining Software Repositories (MSR)*.

Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the self-verification limitations of large language models on reasoning and planning tasks. *arXiv preprint arXiv:2402.08115*.

Riku Takaichi, Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, Toshiyuki Kurabayashi, Hiroyuki Kirinuki, and Haruto Tanno. 2022. Are nlp metrics suitable for evaluating generated code? In *International Conference on Product-Focused Software Process Improvement*, pages 531–537. Springer.

Malik Muhammad Umer. 2025. Comparative analysis of code generated by popular LLMs for MISRA C++ compliance. *arXiv preprint arXiv:2506.23535*.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. [Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation](#). In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Punta Cana. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.

Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2025. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. In *Findings of the Association for Computational Linguistics*.

Armin Zirak and Hadi Hemmati. 2024. Improving automated program repair with domain adaptation. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–43.

A Evaluation Metrics Overview

Choosing what to measure is itself a decision when evaluating safety-critical automotive code, since no single metric captures its quality and the practical options differ in how much tooling they need and how much their verdicts can be trusted. Table 2 surveys the metric families we considered, rating each on tool support, implementation complexity, reliability, and cost. The comparison motivates the choices in Section 4: we rely on standards conformance, which combines high tool support with high reliability, and on support metrics such as latency, while declining reference-based similarity measures whose reliability is only moderate and whose relevance to certification is weak.

B Dataset Preprocessing Pipeline

This appendix expands the pipeline summarized in Section 3.1. The preprocessing pipeline begins by traversing automotive C/C++ repositories to extract individual functions along with their surrounding context, which includes:

- **Function Documentation:** Both Doxygen-style comments (Doxygen, 2025) and inline documentation are extracted and linked to their corresponding function implementations
- **System Requirements:** The pipeline parses DOORS export XML files (IBM, 2025) to map requirement identifiers mentioned in function documentation to their full requirement texts
- **Architectural Context:** PlantUML diagrams (PlantUML, 2025) referenced in documentation are extracted and included to provide behavioral specifications

- **Code API Context:** Related header files and their function signatures are gathered to provide information about available APIs and data structures

Header linking works as follows: the system identifies the main header file corresponding to each C/C++ source file and extracts additional context from included dependencies, so that generated code can reference existing APIs rather than reimplementing them.

The context aggregation process is implemented as follows:

Listing 3: Context aggregation from preprocessing pipeline

```

1 def create_training_example(function_info):
2     user_message = f"Implement the function: {
        ↳ function_info['signature']}\\n\\n"
3
4     if function_info['documentation']:
5         user_message += f"with following
        ↳ specifications:\\n{function_info
        ↳ ['documentation']}"
6
7     if function_info['requirements']:
8         user_message += f"\\n\\nRequirements
        ↳ texts:\\n{function_info['
        ↳ requirements']}"
9
10    if function_info['uml_diagram']:
11        user_message += f"\\n\\nThe behavior
        ↳ follows this UML diagram:\\n{
        ↳ function_info['uml_diagram']}"
12
13    return {
14        "messages": [
15            {"role": "user", "content":
        ↳ user_message},
16            {"role": "assistant", "content":
        ↳ function_info['
        ↳ implementation']}
17        ]
18    }
```

For quality filtering and validation, the preprocessing pipeline implements several quality assurance mechanisms:

- **Function Signature Validation:** Automatically corrects function signatures in imple-

Table 2: Evaluation metrics for automotive code quality assessment. Each category is rated on Tool Support (availability of automated tools), Complexity (implementation effort), Reliability (consistency of results), and Cost (resources required). High = better, except for Cost and Complexity where low is preferable.

Category	Example Metrics	Tool Support	Complexity	Reliability	Cost
ML Static Analysis	BLEU, ROUGE, METEOR	High	Medium	Medium	Low
Static Analysis	Linters, Complexity	High	Low	High	Medium
Dynamic Analysis	Coverage, Runtime	Medium	High	High	High
Standards Compliance	Industry Coding Standards (e.g., MISRA, CERT C)	High	Medium	High	Medium
Support Metrics	Latency, Time	Low	Low	Medium	Low
Business KPIs	ROI, Time-to-Market, Productivity	Low	Low	Low	Medium

mentation files by matching them against header file declarations

- **Documentation Completeness:** Only functions whose documentation specifies the intended behavior are included in the training set
- **Code Compliance:** Functions are validated against AUTOSAR (AUTOSAR, 2017) and MISRA-C (Association, 2013) patterns to assess the initial compliance level

Finally, to ensure balanced representation across different code complexity levels, the preprocessing pipeline implements stratified sampling based on function length and complexity. This approach creates training and test sets that maintain consistent distribution characteristics. The resulting dataset contains approximately 6,000 high-quality function implementations with their complete contextual information, split into training and evaluation sets while maintaining complexity distribution balance.

C Model Architecture and Training Configuration

We select Qwen3-32B as the base model (Yang et al., 2025) for its performance on code generation tasks at a size we could fine-tune on available hardware. Training uses Low-Rank Adaptation (Hu et al., 2021) with rank-8 adapters and $\alpha = 16$, applied to the query, key, value, and output projections together with all feed-forward components, plus 4-bit quantization via BitsAndBytes (Dettmers et al., 2023) to reduce the memory footprint. None of these choices are specific to automotive code: they are a standard parameter-efficient recipe, reported here for reproducibility rather than offered as contributions.

Training runs on Amazon SageMaker with PyTorch DeepSpeed (Rasley et al., 2020), orchestrated across multiple GPUs through SageMaker’s remote function decorator, as shown in Listing 4.

Listing 4: SageMaker remote training configuration

```
1 @remote(
2     instance_type="ml.p4d.24xlarge",
3     volume_size=100,
4     use_torchrun=True,
5     pre_execution_commands=[
6         "pip install torch==2.5.1 transformers=
          ↳ =4.51.3",
7         "pip install peft==0.15.2 deepspeed
          ↳ =bitsandbytes",
8     ]
9 )
10 def train_model(train_dataset, test_dataset,
    ↳ config):
```

```
11 # Training implementation with adaptive
    ↳ DeepSpeed configuration
12 stage = 2 if use_quantization else 3
13 deepspeed_config = {
14     "zero_optimization": {
15         "stage": stage,
16         "overlap_comm": True,
17         "contiguous_gradients": True,
18         "offload_optimizer": {"device": "
          ↳ cpu", "pin_memory": True}
19     }
20 }
```

We select the DeepSpeed ZeRO stage (Rajbhandari et al., 2020) according to whether quantization is enabled. ZeRO-2 partitions optimizer states and gradients while replicating parameters on each device; ZeRO-3 additionally partitions the parameters, which permits larger models at the cost of communication overhead and, for our long prompts, distributes the memory burden of storing full parameters. We default to ZeRO-2 under quantization, since quantized parameters interact poorly with ZeRO-3’s parameter-gathering, and ZeRO-2 also requires fewer all-gather operations. The remaining settings are conventional: gradient accumulation for a larger effective batch size, gradient checkpointing and CPU offloading to fit the model in memory, Flash Attention 2 (Dao et al., 2022; Dao, 2023), and NCCL with communication overlap and extended timeouts. The one detail worth flagging is that our prompts are long, since they carry resolved requirement texts, UML fragments, and header context, so context length rather than model size drove most of our memory pressure.

The final model is packaged through Amazon Bedrock’s custom model import functionality for integration with the multi-model architecture. The distributed setup reaches approximately 85% GPU utilization with stable convergence, which kept each fine-tuning cycle within a development sprint.

D Conversation Format Example

The training data utilizes HuggingFace’s chat template (Hugging Face, 2024) to structure the automotive coding task as a conversational interaction. Listing 5 gives the full, untruncated version of the example abbreviated as Listing 1 in Section 3.1, showing both resolved requirement texts, the complete PlantUML fragment, and the target implementation as it appears in training. This conversational format supplies the model with the requirements, safety level, and behavioral specification alongside the signature, so that the obligations a function must satisfy are present in the prompt rather than inferred from the signature alone.

Listing 5: HuggingFace chat template example for automotive C++ code

```

1  training_example = {
2      "messages": [
3          {
4              "role": "user",
5              "content": ""Implement the
                      ↳ function:
6  void ValidateEngineParameters(const
                      ↳ EngineData_t* engineData,
                      ↳ ValidationResult_t* result)
7
8  with following specifications:
9  /**
10 * @brief Validates engine parameters according
                      ↳ to AUTOSAR specification
11 * @param engineData Pointer to engine data
                      ↳ structure containing RPM, temperature,
                      ↳ pressure
12 * @param result Pointer to validation result
                      ↳ structure
13 * @requirements REQ_ENG_001: Engine RPM must
                      ↳ be within 800-6000 range
14 * @requirements REQ_ENG_002: Engine
                      ↳ temperature must not exceed 110 degrees
                      ↳ C
15 * @safety_level ASIL-D
16 */
17
18 Requirements texts:
19 REQ_ENG_001: The engine control unit shall
                      ↳ validate that engine RPM remains within
                      ↳ the
20 operational range of 800 to 6000 RPM during
                      ↳ normal operation.
21 REQ_ENG_002: The engine control unit shall
                      ↳ monitor engine coolant temperature and
                      ↳ trigger
22 protective measures when temperature exceeds
                      ↳ 110 degrees C.
23
24 The behavior follows this UML diagram:
25 @startuml
26 ValidateEngineParameters -> CheckRPMRange :
                      ↳ validate RPM
27 CheckRPMRange -> CheckTemperature : if RPM
                      ↳ valid
28 CheckTemperature -> SetValidationResult : set
                      ↳ result
29 @enduml"""
30     },
31     {
32         "role": "assistant",
33         "content": ""
34 void ValidateEngineParameters(const
                      ↳ EngineData_t* engineData,
                      ↳ ValidationResult_t* result)
35 {
36     /* MISRA-C 2012 Rule 1.3: No undefined
                      ↳ behavior */
37     if ((engineData == NULL) || (result == NULL
                      ↳ ))
38     {
39         if (result != NULL)
40         {
41             result->isValid = false;
42             result->errorCode =
                      ↳ VALIDATION_NULL_POINTER;
43         }
44         return;
45     }
46
47     /* Initialize result structure */
48     result->isValid = true;
49     result->errorCode = VALIDATION_SUCCESS;
50
51     /* REQ_ENG_001: Validate RPM range (800-
                      ↳ 6000) */
52     if ((engineData->rpm < 800U) || (engineData
                      ↳ ->rpm > 6000U))
53     {
54         result->isValid = false;
55         result->errorCode =
                      ↳ VALIDATION_RPM_OUT_OF_RANGE;
56         return;
57
58
59     /* REQ_ENG_002: Validate temperature limit
                      ↳ (110 degrees C) */
60     if (engineData->temperature > 110.0f)
61     {
62         result->isValid = false;
63         result->errorCode =
                      ↳ VALIDATION_TEMPERATURE_EXCEEDED;
64         return;
65     }
66     /* All validations passed (result pre-
                      ↳ initialized to success) */
67 }"""
68     }
69 ]
70 }

```
