論文・報告

Leveraging AI Tools in Powertrain Engineering Projects

– Rapid Development of MATLAB - Based Windows Applications for Improving Productivity and Accelerating Knowledge Sharing –

Shivansh Singhal*　　Thirumal Harikrishnan*

Abstract

ISUZU has committed to using AI to improve its existing engineering workflows. This paper presents examples of how this is being put into practice, exploring how AI tools can accelerate engineering software development in a commercial vehicle powertrain environment. Two MATLAB-based productivity tools were developed using Generative AI as a development accelerator. Engineers directed the development process by defining requirements and validating outputs, while the AI platform generated code, proposed solutions, and supported debugging. In both cases, fully featured tools were delivered within days rather than the weeks a conventional approach would have required. The results show that AI-assisted development is a practical productivity accelerator for domain engineers, even without dedicated software development resources.

要　旨

いすゞは、開発環境にAIを導入することを推進している。商用車のソフトウエア開発において、AIツールがいかに開発を加速できるか検証を行った。生成AIに要求事項や出力内容を指示し、２つのMATLABツールを作成した。両方のケースで、従来の進め方では数週を要した結果が数日で得られ、AIを活用して開発を進めることで相当な開発期間短縮につながる実用的な結果が得られた。その取り組みを紹介する。

1　Introduction

Modern diesel powertrain development is becoming increasingly complex. Emissions regulations such as Euro 7, Environmental Protection Agency (EPA) 2027, and California Air Resources Board (CARB) Ultra Low Nitrogen Oxides (NOx) are setting targets that are harder than ever to meet. Achieving these targets requires precise control of combustion, aftertreatment, and On-Board Diagnostics (OBD) systems. This means more software functions, more calibration parameters, and more complex signal interactions within the Engine Control Unit (ECU).

OBD requirements are also growing in scope. Engineers must ensure that fault detection and diagnostic strategies are correctly implemented and calibrated across a wide range of operating conditions. Gaps in software understanding can lead to warranty issues in the field, which are expensive and difficult to resolve after production.

As a result, engineers are expected to develop a deep understanding of increasingly complex ECU software in shorter timeframes. The volume of supporting documentation continues to grow. Managing and navigating this documentation has become a significant challenge.

This paper looks at how Artificial Intelligence (AI) tools can help with this challenge, using ChatGPT (OpenAI) Enterprise as a development accelerator. Before going further, it is worth being clear about how this was used. The AI platform was never used to read or process any proprietary engineering documents. Its role was limited to generating the MATLAB (MathWorks) code that engineers then used to process these documents locally, within their own MATLAB environment. The platform itself never had access to the actual document content. All developments took place on ISUZU governed enterprise servers, with enterprise level data protection in place throughout. None of the information used during development

*Isuzu Technical Center of America, Inc.

Copyright© 2026 ISUZU MOTORS LIMITED.
All Rights Reserved. 無断転載・複製・配布禁止

was shared for model training, and no proprietary data ever left the controlled environment.

1.1 Industry Context and Engineering Challenges

ECU calibration projects present a particular challenge within this environment. Before productive calibration work can begin, engineers must develop a thorough understanding of the interlinked functions within the ECU software, a familiarization process that relies heavily on documentation often spanning thousands of pages across multiple file formats. For established programs with well-understood ECU software, this process is supported by accumulated team expertise and documented institutional knowledge.

However, new ECU programs introduce a substantially greater burden, engineers must independently trace internal signal flows, decode function hierarchies, and map software interdependencies with limited prior reference. In the absence of structured tooling to support this process, the familiarization phase alone can consume a substantial portion of engineering time, time that would otherwise be directed toward productive calibration and validation activity. **Figure 1** illustrates the typical software familiarization workflow within such programs highlighting the specific stages where knowledge acquisition bottlenecks are most acute.

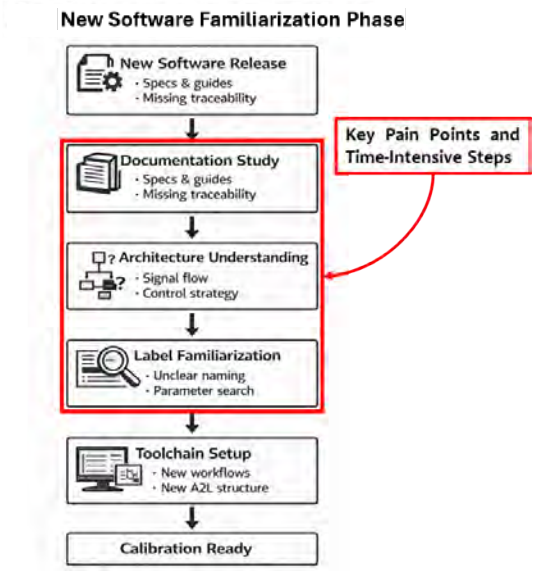


Figure 1 Structured Approach to Software Familiarization

1.2 Gaps in Current Practice

Model-based development has brought structure to many parts of the engineering process. However, when working with supplier-developed ECU software, teams often have limited visibility into the internal software architecture. The documentation provided varies in format, depth, and quality across suppliers and programs. There is no standard way to navigate it, and the burden of understanding falls largely on individual engineers.

In practice, teams rely on manual document review, informal knowledge sharing between colleagues, and ad hoc search methods. These approaches are slow. They also produce inconsistent results. One engineer may develop a strong understanding of a particular software function while a colleague working on the same program has very little. This inconsistency creates risk. Decisions made on incomplete understanding of the software can lead to calibration errors, missed OBD requirements, or warranty issues that only surface in the field.

The challenge is made worse when engineers move between programs or when a new ECU platform is introduced. The knowledge built up on one program does not transfer easily to the next. Each transition requires a significant investment of time before productive calibration work can begin. For teams working under tight program timelines, this is a recurring and costly problem.

There is a clear need for tools that help engineers access and understand software documentation more efficiently. This paper presents two such tools, developed using AI-assisted methods within a commercial vehicle powertrain environment. **Section 3** covers the development process and engineering impact of each tool in detail.

2 Workflow Development Framework

2.1 Session Initialization and Context Setting

The AI platform used in this effort is the ChatGPT enterprise tool. Each development session began with deliberate context establishment. The engineer configured the session by first orienting the platform toward the specific development environment. A typical opening prompt would resemble: "*We will be working in MATLAB throughout this session. Please refer to MATLAB syntax, functions, and documentation*

conventions in all responses." This was followed by a description of the engineering problem – what the tool needed to do, what the input data looked like, and what the expected output should be. For example: "*I have a folder of Word and Portable Document Format (PDF) documents containing ECU software information. The files are a mix of tables and free text. I need to extract calibration label names, input and output signals, min/max values, units, and which software feature each label belongs to.*"

Critically, the engineer did not immediately request code. Instead, the platform was asked to suggest possible approaches first: "*Before writing any code, can you suggest two or three ways we could approach parsing these documents in MATLAB? I want to understand the options before deciding.*" The platform would then outline candidate strategies, and the engineer would evaluate these against the engineering requirements before selecting a preferred direction. Only once a satisfactory approach was agreed did the engineer provide a detailed output specification, forming the basis for the first code generation pass. From this point, development proceeded through a continuous cycle of prompt and response.

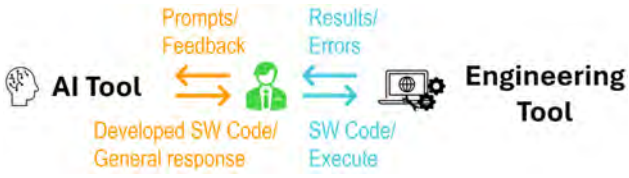


Figure 2 Engineer-Mediated Iteration Cycle Between AI Tool (ChatGPT) and Engineering Tool (MATLAB)

2.2 Project-Based Continuity

Development was organized using the Projects feature within ChatGPT, which allowed multiple related chat sessions to be grouped under a single project context. This preserved accumulated knowledge (context for the Large Language Model (LLM)) across sessions, enabling the model to maintain awareness of previously developed components, design decisions, and established conventions without requiring the engineer to re-establish context at the start of each session. This continuity was particularly valuable across the multi-phase development lifecycles of both tools, where later sessions were built directly upon earlier architectural decisions.

2.3 The Iteration Cycle

There was no direct interface between the AI platform and the MATLAB environment during development. The platform was prompted to write specific aspects of the code, which the engineer then manually copied into MATLAB and executed. Feedback was returned to the platform in two ways. The first was text-based MATLAB error messages pasted directly into the chat, or qualitative observations about the output were described in words. For example, when the parsing script produced incomplete results, the engineer would return to the session with a prompt such as: "*The script runs without errors but is missing labels that appear in paragraph text rather than tables. Here is the error output: [pasted error]. Can you update the parsing logic to also capture these?*"

The second feedback method was image-based. Where the issue was visual in nature, such as incorrect application layout, unexpected User Interface (UI) behavior, or output formatting problems. In these cases, screenshots of the MATLAB environment were shared directly in the chat. The platform was able to interpret these images, identify the shortcoming from the visual context, and suggest corrective code without requiring the engineer to describe the problem in words. This proved particularly useful for debugging UI components and output formatting, where a screenshot communicated the issue far more precisely than a text description could.

The platform reworked the code based on both types of feedback, and the cycle repeated. Initial functional prototypes were typically achieved within a small number of exchanges. Subsequent iterations were driven by fine-tuning and edge case handling rather than fundamental correctness issues. All refinements were handled within the AI tool rather than engineer directly editing the generated code, hence preserving a clean and traceable development dialogue.

2.4 Engineer's Role and Validation

Engineer brought an intermediate level of MATLAB proficiency to the collaboration, sufficient to evaluate the correctness and structure of generated code, identify logical errors, and direct refinement without being dependent on AI generation for every aspect of implementation. Technical validation of the developed tools was conducted jointly by the engineer and colleagues, ensuring that outputs were assessed

against real engineering requirements. This validation approach was essential in identifying edge cases. Such cases were addressed by explicitly surfacing them within the AI chat session, enabling the model to incorporate appropriate-handling logic in subsequent code revisions.

Figure 2 Illustrates how the engineer bridged communication between the AI Tool and Engineering Tool.

2.5 Methodology Maturation

Prompting practice evolved considerably across the project. Early sessions used broad, exploratory prompts. For example: "*Can you write a MATLAB script to extract data from these documents?*" Over time, prompts became more precise and specific. For example: "*Write a MATLAB function called parseWordTables. It should take a file path as input, extract all table rows where the first column matches the pattern 'XYZ', and return a structured array with fields: labelName, minValue, maxValue, unit, and featureName.*"*

This shift came from a growing ability to specify requirements clearly and anticipate edge cases upfront. The engineer also became better at recognizing early when the platform was going in the wrong direction, allowing quick correction before significant rework was needed. This improvement in prompting effectiveness directly contributed to the faster development observed in later phases. It also represents a transferable skill that can be applied to future AI-assisted engineering work.

3 AI-Assisted MATLAB Tool Development: Powertrain Engineering Case Studies

3.1 Case Study 1: Software Documentation Rapid Search Toolchain

Of the familiarization challenges outlined in **Section 1**, navigating large volumes of calibration documentation was the most time-intensive bottleneck in day-to-day engineering work. Engineers had to manually search across Word and PDF files spanning thousands of pages. There was no structured way to query, filter, or cross-reference signal data. Most content was in structured tables, but some appeared as free text, with both formats spread across multiple pages within the same files. This made the process slow, inconsistent, and heavily dependent on individual familiarity with the documents.

The tool was built across three phases: document parsing, database creation, and application delivery, as illustrated in **Figure 3**.

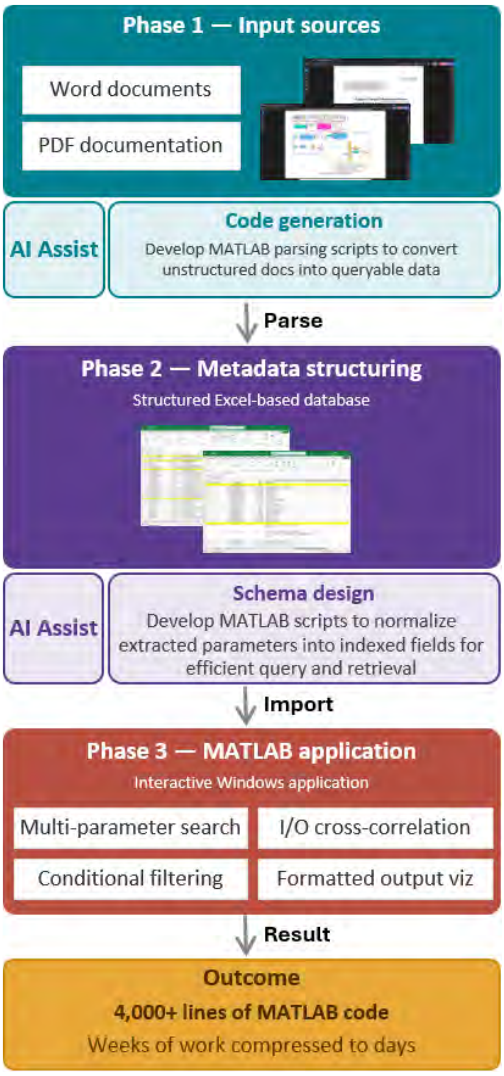


Figure 3 Layered pipeline architecture of the Software Documentation Rapid Search Toolchain, illustrating phase-wise integration of AI-assisted development and resultant application capabilities

Development started with the engineer describing the problem to the platform. The documents were a mixed set of Word and PDF files, mostly in table format but with some free-text sections, spread across multiple pages. The engineer also described what data needed to be extracted: calibration label names, input and output signal definitions, boundary values, units, functional descriptions, and software feature attribution.

Rather than jumping straight to code, the engineer first asked the platform to suggest possible parsing approaches. The platform outlined several options. The engineer then selected the most suitable one, which used separate routines for table content and free-text paragraphs. This early discussion shaped the direction of the entire development. As the work progressed, the engineer also asked the platform to add comments throughout the generated code. This made it easier to follow the logic, understand each step, and continue development across multiple sessions.

Second phase was about storing the extracted data in a format that could be easily searched and queried. Engineer specified to the AI tool on what the database should look like "*Three separate Excel worksheets for calibration labels, input signals, and output signals, each capturing parameter-level metadata*". The AI tool generated the Excel structure and the MATLAB code to write data into it.

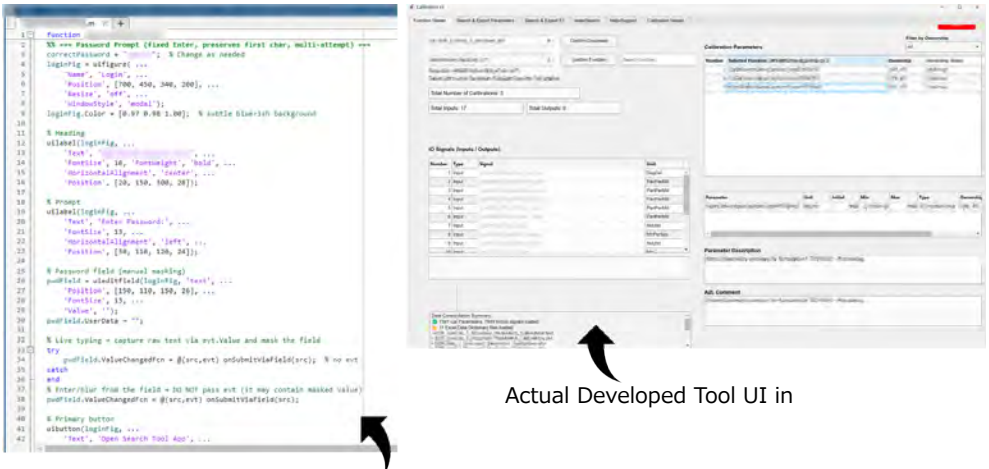
During testing, a problem became apparent. The extraction was picking up a large number of uncertain or incomplete records alongside the valid ones. The engineer described this to the AI tool, which suggested adding a filtering step. High confidence records would go into the main sheets, and uncertain ones would be moved to a separate review sheet for manual inspection. This change improved the quality of the database significantly. A separate issue also came up during testing: a MATLAB error related to column naming. The engineer pasted the error into the chat, and the AI tool returned a corrected script straight away.

Third phase was the application itself. A Windows desktop application was built using MATLAB's uifigure framework. The engineer described what the application needed to do, one requirement at a time. These included keyword search, exact field lookup, approximate query modes, signal cross-correlation analysis, conditional filtering, and formatted output for engineering review. The application was made available in two ways: as a standalone executable for engineers, and as a native MATLAB application.

The codebase was organized as one main script with a set of helper functions. Each helper function handled a specific task such as parsing, database writing, search, or UI rendering. New features were added one at a time by requesting new helper functions from the platform and connecting them to the main script. The engineer also asked the AI tool to include comments throughout the generated code. This made it easier for colleagues to understand the code and made it simpler to continue development across multiple sessions.

It is worth noting that the AI tool was not used to process any proprietary documents. It was only used to help build the tool itself. All document data was handled within internal engineering systems only.

The first working prototype was ready within hours of starting development. The complete application, with over 4,000 lines of MATLAB code, was delivered within days. A conventional development approach would have taken several weeks. **Figure 4** shows the finished application interface and a sample of the generated MATLAB code.



MATLAB Code to Implement the Tool UI and Integrate All the Information into a Single, User-FriendlyIn terface

Actual Developed Tool UI in

Figure 4　Realized MATLAB Application Interface and Supporting Code Structure

3.2 Case Study 2: Automated Simulink Model Generation Toolchain

Building Simulink models manually is a time-consuming process. For each software function, the engineer has to create subsystem blocks and populate them with the correct inputs, outputs, calibration labels, and lookup tables by hand. This information has to be read from specification documents and entered into Simulink one parameter at a time. The process is also prone to error. A missed signal or incorrectly named block can cause problems later in the development cycle that are difficult to trace back. For the software and controls team, this setup work took up a significant amount of time before any real modelling could begin. It also had to be repeated from scratch for every new software function.

The tool was developed in MATLAB using the same workflow described in **Section 2**. As shown in **Figure 5**, the process has two parts. The first part is AI-assisted, where the AI tool helped develop the MATLAB scripts to read and process the specification data. The second part is engineer-led, where the generated output is reviewed and built upon to create the complete Simulink model. This division kept the engineer in control of the technical decisions while the platform handled the time-consuming code development work.

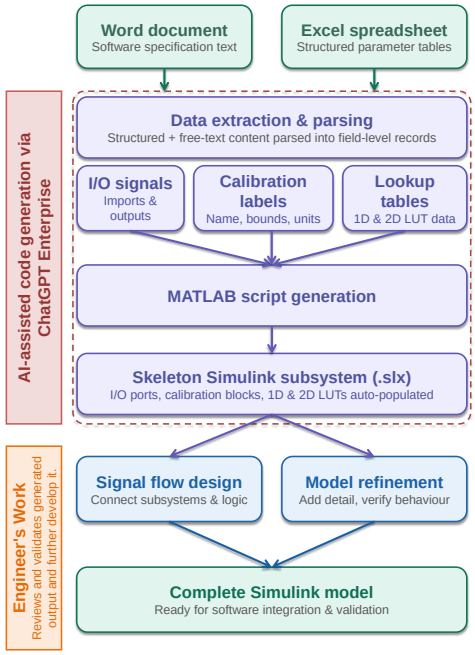


Figure 5 End-to-end workflow of the Automated Simulink Model Generation Toolchain – from source documentation to complete Simulink subsystem

Development started with the engineer describing the problem to the platform:

"*I have Word and Excel documents describing ECU software functions. Each document defines the input signals, output signals, calibration labels, and lookup tables for a function. I want to write a MATLAB script that reads this information and automatically creates a Simulink subsystem block with the correct Input/Output (I/O) ports, calibration blocks, and lookup tables pre-populated.*"

The platform asked a few clarifying questions about the document structure before suggesting an approach. For example, it asked how the signal names and lookup table data were laid out in the Excel sheets, and whether the Word documents followed a consistent format across all functions. This helped make sure both the engineer and the platform had the same understanding of the problem before any code was written. It also helped avoid rework later in the development.

The tool read specification data from both Word and Excel files. These were the two formats used to store function definitions, signal lists, calibration parameters, and lookup table data across the project. Once the basic parsing logic was working, the engineer refined the requirements through follow-up prompts as new cases came up during testing. For example:

"*The lookup tables can be either 1D or 2D. Can you update the script to handle both cases?*"

Each follow-up prompt added a new capability to the script. This is a good example of how the tool grew incrementally through the development process rather than being fully specified at the start. The final scripts generated skeleton Simulink subsystem blocks directly from the source documents. Each block included the correct input and output ports, calibration label blocks, and 1D or 2D lookup table blocks where needed. Block names were taken directly from the document content, keeping the generated model consistent with the original specification.

The engineer's role, shown in the lower part of **Figure 5**, was to review the generated blocks and then build on them. The blocks were used mostly as they were. The engineer then connected them, defined the signal flow between functions, and added the logic needed to complete the software architecture. The auto-generated blocks gave the engineer a reliable starting point and removed the need to set up each subsystem from scratch.

The main benefit was removing the repetitive manual setup work. Previously, an engineer had to locate each parameter in the specification document and enter it into Simulink individually. For a function with many signals and multiple lookup tables, this could take a considerable amount of time. The tool did this automatically and consistently. **Figure 6** shows a snippet of the generated Simulink Wireframe and a sample of the generated MATLAB code. The tool removed a time-consuming step from the workflow, allowing engineers to start signal flow design and software logic implementation sooner. The benefit was most noticeable for complex functions with large signal interfaces. The tool was initially used by the software and controls team, but the approach showed clear potential for wider use across model-based development workflows.

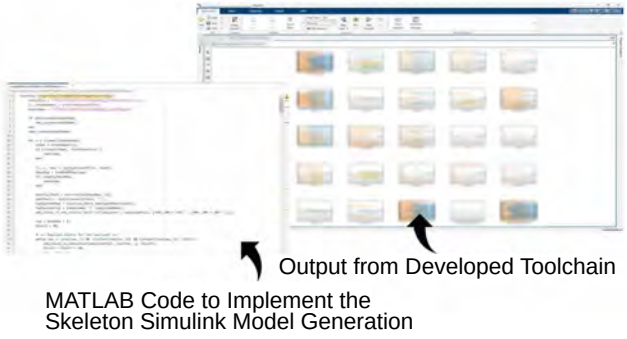


Figure 6 Realized MATLAB Code for Automated Simulink Model Generation and its output

4 Future Scope for Organizational Workflow Enhancement

The tools presented in this paper address specific workflow bottlenecks. However, the underlying development approach is transferable to any engineering activity that involves repetitive tasks, large volumes of data, or structured information processing. The organizational opportunity is therefore considerably broader than the individual case studies suggest.

Within engineering organizations, the most immediate gains are likely in workflows where the inputs and outputs are well defined, and the tasks are time-consuming but not highly creative. These are actly the conditions where AI-assisted development provides the largest productivity boost – freeing up engineers to work on higher-value problem solving and decision making.

As illustrated in **Figure 7**, looking further into the future, the maturation of agentic AI systems represents a more fundamental change. Unlike the assistive approach shown in this paper – where a human engineer stays in the loop at every step, agentic systems can autonomously plan and execute multi-step workflows, work on tools, make intermediate decisions, and adapt based on results. As these capabilities mature, the line between human-directed and AI-directed engineering work will continue to shift.

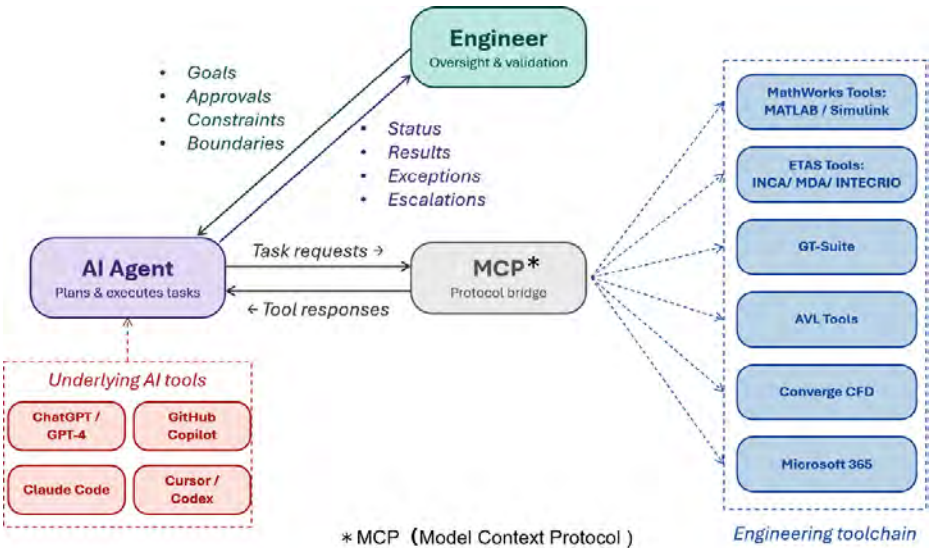


Figure 7 Agentic AI workflow for powertrain engineering: AI agent orchestrating multi-tool execution via MCP (Model Context Protocol) under engineer oversight

5 Conclusion

The case studies presented in this paper demonstrate that generative AI can fundamentally alter the economics of internal toolchain development, compressing what would traditionally require weeks of specialist effort into days, without compromising engineering rigor or data governance.

Beyond development speed, AI-assisted development expanded the scope of what engineers considered feasible to build. Problems that would previously have been worked around due to development time constraints could now be directly addressed through targeted tooling, shifting the engineering mindset from adapting to limitations toward actively resolving them.

Perhaps most significantly, this work demonstrates that effective AI-assisted development is itself a learnable and transferable skill. Engineers can achieve productivity benefits that go far beyond any one tool or project by investing in creating collaborative workflows with AI systems and organized prompting practices.

あとがき

この取り組みを開始した当初は、それほど多くの成果を期待していませんでした。しかし、すぐにこのプラットフォームがエンジニアリング的な文章を驚くべきレベルで理解可能であることが明らかになりました。通常であれば作成に何時間もかかるような膨大で複雑な成果物をわずか数秒で生成し、初回の試行にも関わらず精度が70%を大きく超えていました。この事実は、1日の業務中に何を達成できるかという常識を根本から覆す可能性を秘めています。

とはいえ完全にシステム任せにできるわけではありません。プラットフォームは度々当初の目標から逸脱し、一見正しいように見えても実際の問題を完全には解決していないアウトプットをあたかも正しいかのように生成することがあります。そのためエンジニアは常に主導権を握り、プラットフォームを適切にコントロールし続ける必要があります。

最後に、本稿を通じて単なるツールの紹介にとどまらないメッセージをお伝えできれば幸いです。AIを慎重な検討と明確な線引きのもとで活用すれば、エンジニアの能力と生産性を高めることができると私たちは確信しています。ここでご紹介した取り組みは、チームやプロジェクト、そして私たちが開発した車両を信頼してくださるお客様に、高付加価値をお届けするより良い方法を絶えず追求し続けるという、ITCA(Isuzu Technical Center of America Inc.)の姿勢を体現するものです。

(ITCA Powertrain Engineering　岩下拓郎)