AI-Assisted Porting of Automotive System Software to New Hardware Architectures

Andreas Kappel*
Pro2Future GmbH
Graz, Austria
andreas.kappel@pro2future.at

Aimen Medjek*
Pro2Future GmbH
Graz, Austria
aimen.medjek@pro2future.at

Florian Riedl*
Graz University of Technology
Graz, Austria
florian.riedl@tugraz.at

Amer Kajmaković
Pro2Future GmbH
Graz, Austria
amer.kajmakovic@pro2future.at

Tobias Scheipel
Graz University of Technology
Graz, Austria
tobias.scheipel@tugraz.at

Marcel Baunach
Graz University of Technology
Graz, Austria
baunach@tugraz.at

Abstract

The manual porting of software to new hardware is a time-consuming and error-prone process, especially in the embedded automotive domain where safety, correctness, and traceability are essential. This is even more true when the target architecture differs significantly from the origin architecture, requiring us to rethink various low-level abstractions. This paper outlines a research vision for transitioning to an AI-assisted porting workflow that respects the unique requirements imposed by automotive standards like AUTOSAR and MISRA C. We first combine our findings on manual porting into a single workflow, which identifies six steps we aim to augment and eventually automate using Agentic Artificial Intelligence. We then propose using Knowledge Graphs to consolidate all porting-relevant information while supporting traceability, which standards such as ISO 26262 often mandate for automotive software. Finally, we outline our approach to addressing specific challenges of understanding embedded systems source code, such as assembly code and preprocessor directives.

1 Introduction

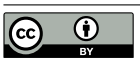
Embedded systems [30, 36] have become the invisible computational backbone of modern technology, ranging from smartphones and household appliances to medical devices, industrial automation, and automotive systems. The continued growth of the global embedded systems market highlights the increasing demand for embedded technologies across diverse application domains [11].

These embedded systems combine software and hardware to fulfill a particular task [15]. This dependency is particularly pronounced in safety-critical systems, such as those used in the automotive domain. In this context, software or hardware failures can have catastrophic consequences, including threats to human life, substantial environmental damage, and significant financial losses [24].

As mentioned by Martins Gomes et al. [23], embedded Operating System (OS) developers face the challenge of ensuring support for a wide range of heterogeneous target architectures. Conventionally, porting software stacks across diverse hardware platforms has been a time-consuming and error-prone manual task. Especially low-level code sections, which directly interact with the hardware, such as interrupt handling, system initialization, or context switching, often require architecture-specific modifications or reimplementation. This task requires in-depth knowledge and expertise about the hardware, the software, and their interaction.

This hardware-software co-dependency and the need for frequent porting pose significant challenges for software development in modern vehicles, which run millions of lines of code distributed across numerous Electronic Control Units (ECUs) [5, 6]. Historically isolated, mechatronic components have evolved into integrated, distributed networks where previously independent functions, such as braking, steering, and engine control, must actively interact. This dispersion of functionality across a complex, distributed infrastructure introduces substantial risks of unintended interactions between features. Furthermore, the automotive industry operates under intense competitive and cost pressures, which often drive software optimization to the limits of the available hardware [6, 14]. As a result, software is typically highly tailored to specific hardware architectures, making reuse, adaptation, and porting considerably complex. Consequently, developers may need to reimplement and adapt software repeatedly to accommodate new hardware platforms.

*Authors contributed equally to this research.



This work is licensed under a Creative Commons Attribution 4.0 International License. DOI: https://dx.doi.org/10.18420/fgbs2026h-02. GI/ITG FG Operating Systems, October 08–09, 2026 Erlangen, Germany.

The increasing complexity and effort of manual software porting therefore motivate our proposal for intelligent methods that support and automate this process. Szymanski et al. [31] emphasize on leveraging Large Language Models (LLMs) for complex knowledge-based use cases across various industries. In addition, Brohi et al. [4] add that both Agentic Artificial Intelligence (AAI) and LLMs are improving the understanding and generation of language with far-reaching consequences for research and industry practices, decision-making, and task automation. In software engineering, Zhang et al. [34] highlight how LLMs are being deployed to automate a broad range of activities across the software development and maintenance lifecycles. Specifically, their survey highlights the prominent role of LLMs in code translation, and their emerging application to automated code porting.

In this work-in-progress paper, conducted as part of a research project with industry partners, we investigate using Artificial Intelligence (AI) to support porting existing Basic Software (BSW) stacks from an origin to a target hardware platform. The envisioned procedure leverages the existing implementation, hardware and software documentation, and relevant requirement specifications to facilitate and automate porting while preserving functionality and safety-critical constraints. The following sections are organized as follows. Section 2 introduces the background, while Section 3 reviews the related work. Our research vision is given in Section 4, and an outlook is provided in Section 5.

2 Background

Porting automotive BSW from an origin to a target hardware architecture requires adapting hardware-dependent software components. Before introducing the proposed methodology, this section describes the relevant background.

2.1 Basic Software Layered Architecture

The AUTomotive Open System ARchitecture (AUTOSAR) [2] separates the functionality of an ECU into standardized layers, ranging from the Application Layer at the top, over the Runtime Environment and the BSW, down to the actual Microcontroller Unit (MCU) hardware as illustrated in Figure 1, with defined interfaces between them. In this structure, the BSW includes the hardware-dependent part of the stack, since it contains the Microcontroller Abstraction Layer (MCAL) drivers, the lowest layer of the BSW, enabling direct interaction with the hardware and providing an MCU-independent interface to the upper layers, then the ECU abstraction layer uses this interface to define ECU drivers and provides an ECU-independent interface. On top of these layers among the BSW, the Services Layer employs the interfaces from the underlying sub-layers to build the system-management services (e.g. operating-system functionality,

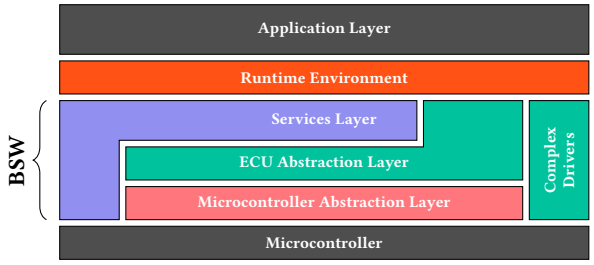


Figure 1: AUTOSAR Layered Software Architecture [1].

communication and network management, memory handling) to the upper layers. Finally, *Complex Device Drivers* are additional drivers which bypass the strict layering and may also directly access the hardware. For this paper, the BSW is the relevant layer for porting to new hardware architectures, while the application layer can remain unchanged [1, 3].

2.2 Software Porting

According to Mooney [26], the term portability in the context of software engineering denotes the capacity of a software to be transferred to another environment. Software is considered portable if porting costs less than redeveloping it from scratch. If a software fully meets this requirement, it is considered completely portable [27]. In general, software portability distinguishes between binary porting, which concerns executable software, and source porting, which facilitates adaptation to a wider variety of target environments [26]. In our research project, we focus on source porting.

2.3 Knowledge Graphs

Knowledge Graphs (KGs) represent domain entities and their interrelations in a networked graph structure [9], making them well suited for capturing the heterogeneous, cross-domain information required for porting automotive software, such as requirements, hardware specifications, and source code artifacts. Code Property Graphs (CPGs) combine abstract syntax trees, control-flow graphs, and program-dependence graphs into a unified representation for static code analysis [33], and can be embedded within a KG to link fine-grained code structure to the surrounding requirement and hardware context.

3 Related Work

De Souza and Castro [7] address the inconsistencies of traditional “trial and error” OS porting by proposing a systematic five-step method for adapting embedded OS to new hardware platforms. Their framework utilizes precedence and dependency matrices to identify all necessary source code modifications. A key contribution is a specialized algorithm

that resolves cyclic references by establishing a clear priority order and an effort-based schedule for porting activities. Unlike the general software development models mentioned in the paper, this approach targets the OS adjustment phases, ensuring that all hardware-dependent source code is identified and modified.

In addition, to automate the migration of legacy code bases to the OpenHarmony platform (an open-source, distributed OS designed for the Internet of Everything), Zhou et al. [35] introduce `ArkAdapter`, an automated, LLM-based code adaption framework designed to port TypeScript and JavaScript (TS/JS) software libraries to ArkTS [17]. `ArkAdapter` implements a Retrieval-Augmented Generation (RAG) pipeline backed by an adaptation knowledge repository containing empirically enriched, few-shot adaptation examples. To handle interdependent syntax mismatches and reduce iterative adaptation loops, `ArkAdapter` employs an adaptation priority strategy based on code dependency structures and syntactic granularity. Evaluated on a benchmark of 209 file pairs, `ArkAdapter` achieved a precision of 86.84% and a recall of 100.0% in locating syntax mismatches. Additionally, it ported 70 of 79 widely used TS/JS libraries to the OpenHarmony Package Manager central repository, passing official compilation and testing verification.

Beyond LLM-based code adaptation, prior work explores KGs, CPG, and Graph Retrieval-Augmented Generations (GraphRAGs) representations for software analysis and porting, as detailed in Section 4.3. Wu et al. [32] use CPGs to port vulnerability patches across code branches, slicing the CPG around a patch before an LLM generates the ported version. Krishnan et al. [21] build a KG from legacy software and related artifacts to support incremental modernization analysis. Min et al. [25] apply GraphRAG to legacy code migration and report up to 15% improvement over plain RAG. Manjunath et al. [22] store micro-architectural knowledge for specialized RAG retrieval and compare it with simple RAG and GraphRAG approaches. Together, this prior work shows that structured knowledge representation can improve the quality and efficiency over classical RAG approaches. However, no prior work has investigated the benefits of a graph-based representation for traceability.

4 Research Vision

This section presents our research vision for Agentic Artificial Intelligence-assisted BSW porting: Section 4.1 introduces the manual porting workflow that we aim to automate, Section 4.2 shows how a Knowledge Graph can be used to represent the relevant requirements, source code, and hardware specifications needed to support each porting step, and Section 4.3 outlines a plan to deal with the special challenges of analyzing embedded automotive source code.

4.1 AI-Assisted Porting Methodology

Embedded software porting is the process of adapting or rewriting existing code and configurations to run on a new hardware platform. This typically involves modifying or reimplementing low-level components, such as device drivers. The primary objective is to satisfy mandatory functional requirements, while performance optimizations may be addressed where feasible as optional aspects of the porting process [16, 26, 28]. In other words, porting embedded software involves adapting and reusing it on a target system for which it was not originally designed or implemented.

De Souza and Castro [7] and the embedded software development company Pebble Bay Consulting [29] independently analyzed the embedded systems software porting workflow and identified the tasks required for a successful porting process. We consolidate and harmonize these findings with insights from discussions with experts at our industrial project partners, as well as our own experience manually porting code. From this, we derive a single manual workflow that serves as the methodological foundation of our AAI vision and consists of the following steps:

(1) **Requirements and Scope Analysis**: Establish a comprehensive understanding of the technical and economic objectives, including business constraints and technical dependencies, along with measures to guide and assess the porting process.
(2) **Collect Origin and Target Information**: Consolidate all relevant project assets, including the origin source implementation and toolchain information, origin and target hardware documentation such as manuals and technical references, and test specifications, together with other supporting materials.
(3) **Hardware Variation Analysis**: Identify hardware-related discrepancies between the origin and target architectures that may affect software portability. This includes software requirements not natively supported by the target platform, such as missing peripherals, different interrupt mechanisms or memory organization, or other board-specific services.
(4) **Dependency and Impact Analysis**: Analyze how the hardware variations identified in `Step 3` influence the existing software implementation. Determine whether each affected software element can be reused as-is, requires adaptation to the target platform, or must be replaced or newly implemented. In addition, identify dependencies and interrelationships between the affected software elements and their associated modifications.
(5) **Port Planning**: Define the porting strategy and sequence of modifications by considering a priori specifications, representing constraints known before the porting analysis, and analysis-derived constraints, emerging from

`Step 3` and `Step 4`. Consolidate these specifications into a prioritized plan that defines priorities, the sequence of modifications, milestones, and risk points.

(6) **Implement the Port**: Implement the port incrementally, starting with the low-level platform foundation and subsequently integrating the required drivers, middleware components, and application-level functionality as needed.

In addition to the six defined steps, documentation and verification are applied throughout each step to ensure that the results, decisions, and artifacts remain consistent with the defined requirements, specifications, and objectives, ensuring correctness, completeness, and traceability.

Figure 2 illustrates our proposed AAI porting workflow, which is based on the conventional manual porting workflow. The conventional porting workflow comprises six steps, numbered 1 to 6 and highlighted in green, and currently involves substantial human intervention and domain expertise. Our goal is to progressively augment and automate individual activities using AI, initially assisting engineers with specific tasks and gradually replacing manual activities as AI capabilities advance. Ultimately, we aim to develop an AAI system capable of autonomously performing the entire porting process for embedded systems while collaborating with human experts when necessary.

The core component of this framework is the `Orchestration Agent`, which serves as the central manager for determining what needs to be done, which agent or human should perform which task, the order in which tasks should be executed, and how their results should be combined. Each step of the manual workflow can be operationalized as an autonomous agent, enabling context-aware decision-making and adaptive task execution. Depending on the complexity of a given task, an agent may be supported by zero to n specialized sub-agents. For example, in `Step 6`, the agent responsible for implementing the port may coordinate multiple specialized sub-agents to perform tasks such as code-template generation or code generation for specific drivers based on the given Software Requirements Specification (SRS). Supporting these agents, the `Data Management Agent` ensures that accurate, relevant, accessible, and governed data are available at appropriate time to support reliable decision-making and task execution throughout the entire workflow. Complementing this, the `Verification Agent` ensures that intermediate and final outputs are correct, consistent and satisfy the relevant requirements and specifications before a task is considered complete. Similarly, the `Documentation Agent` ensures that all decisions, artifacts, and results generated throughout the workflow are accurately captured, traceable, and organized to support transparency and future reuse.

Our long-term vision is to establish an automated AAI framework for the porting of embedded software. We will pursue this objective systematically by incrementally automating the individual steps in the manual porting workflow. Currently, our work focuses on developing specialized agents that support `Step 2` and `Step 6` of this workflow. For `Step 2`, we investigate the automated extraction of relevant information about the origin and target platforms through static code analysis and hardware documentation analysis. In `Step 6`, we focus on generating target architecture code templates based on AUTOSAR, SRS, and MCAL requirements. These initial efforts lay the foundation for progressively increasing automation across the porting workflow.

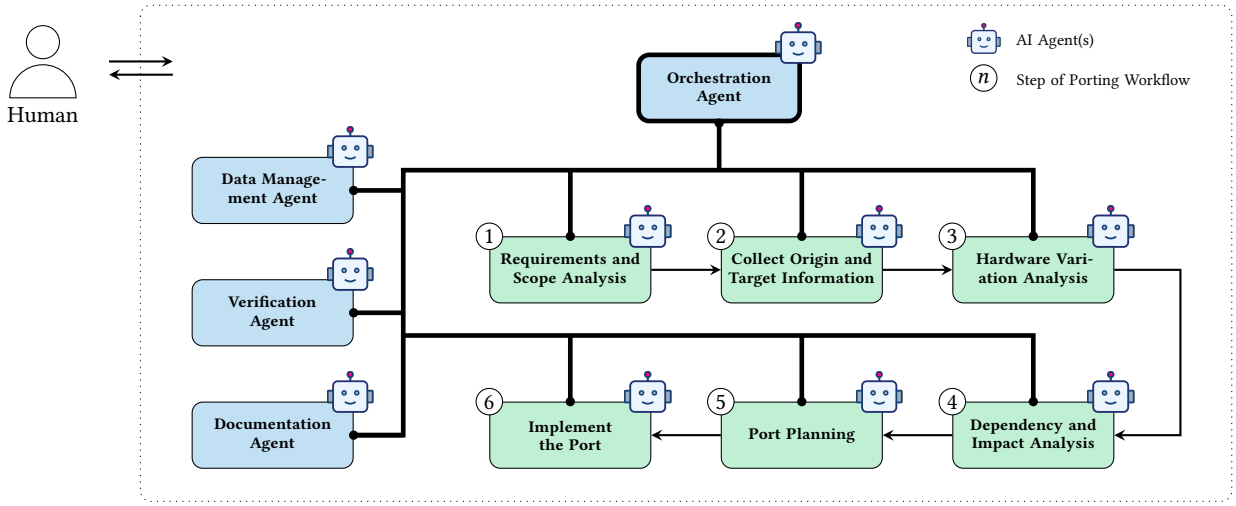


Figure 2: AAI Porting Workflow.

4.2 Data Ingestion and Traceability

To generate a code template for the BSW porting workflow, defined in Section 4.1, we first need to ingest, pre-process, and manage different project artifacts. Relevant information is distributed across these artifacts, from AUTOSAR and MCAL requirements and specifications to Motor Industry Software Reliability Association (MISRA) C guidelines, hardware documentation, and the origin BSW implementation. Establishing and maintaining traceability between these artifacts is essential to identify hardware-dependent components, analyze their dependencies, and support their adaptation from the origin to the target platform.

The IEEE Guide to Software Requirements Specification [18] defined "an SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation". This traceability is further split into backward traceability (referencing each requirement's source in earlier documents) and forward traceability (ensuring each requirement carries a unique identifier so it can be traced to all documents it spawns) [13].

To build an AAI porting pipeline, as introduced in Section 4.1, continuous tracing and querying of requirements is needed. This calls for a database that captures all BSW requirements for each architecture specification.

However, in software development generally, and in embedded development and porting specifically, traditional (relational) databases hit limitations when identifying and querying dense [20], cross-cutting relationships between software and hardware requirements across the two domains. This is where we turn to Graph Databases (GraphDBs), which represents entities (requirements, modules, hardware components, etc.) as nodes and their dependencies as edges, enabling native modeling and querying of relationships that relational schemas struggle to express. Modeling the SRSs as a GraphDB makes it easier for LLMs to ingest, compared with raw documentation.

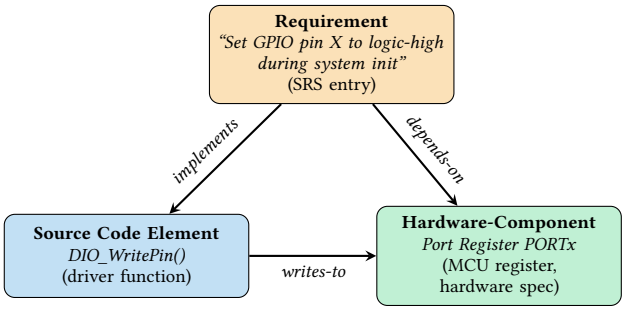


Figure 3: Knowledge graph subgraph for the GPIO pin-initialization requirement example.

These nodes and edges can be represented as a KG by imposing a schema on top of the GraphDB, defining typed entities (e.g., requirements, hardware components, source code elements) and typed relations between them (e.g., implements, depends-on, conflicts-with) [9]; this schema is what enables semantic querying and inference across domains, rather than merely storing linked records. For instance, Figure 3 shows a requirement demanding a specific General Purpose Input/Output (GPIO) pin to be set to a defined logic level during system initialization. This requirement, sourced from an SRS entry, is represented as a **requirement** node, linked via an *implements* edge to a **source code element** node (e.g., the driver function writing to the pin's output register), and via a *depends-on* edge to a **hardware-component** node representing the target MCU's port register, as defined in the hardware specification.

This mapping directly supports the porting workflow in Section 4.1. In `Step 2`, we pre-process all origin and target materials, including the SRS entry, driver code, and both hardware specifications, to construct the GraphDB. In `Step 3`, we query the hardware-specification subgraphs to retrieve the corresponding port register node and analyze the hardware gap between origin and target (e.g., a differing register address or reset value). In `Step 4`, dependency analysis between software and hardware requires querying the CPG subgraph together with the hardware-specification subgraph, linked through the source code-to-hardware edges (e.g., the driver's register access linked to the target port register).

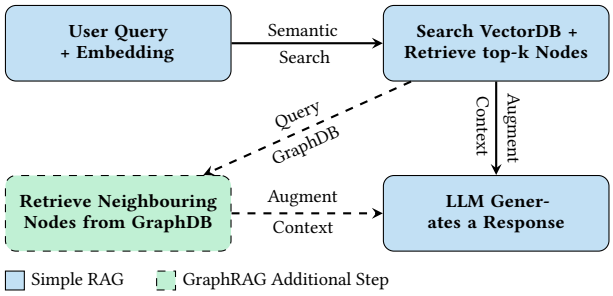


Figure 4: Overview of RAG and GraphRAG.

To employ the GraphDB and make use of its relationships across multiple hardware and software requirements, we opted to use GraphRAG instead of simple RAG. Compared to simple RAG, GraphRAG contains an additional step before the generation part, where we include adjacent nodes to the retrieved ones simply by querying GraphDB starting from the retrieved nodes from the semantic search [8]. For the pin-initialization requirement above, a semantic search first retrieves the requirement node itself; the additional GraphRAG step then pulls in adjacent source code and hardware-component nodes, ensuring the generation step sees the driver

logic and target register definition together rather than the requirement text alone. This improves traceability of requirements and specifications and, as shown in Figure 4, provides better context awareness for the augmented generation step.

4.3 Embedded System Software Analysis

Code Property Graphs play an essential role in our porting methodology, as they allow efficient querying of our origin software for code properties necessary to implement the port [33]. However, existing tools for creating CPGs do not fully support assembly code and preprocessor directives. To faithfully represent the origin source code in our knowledge graph, we plan to develop specialized tools to extract factual information from the provided low-level system code.

Analyzing assembly code. Architecture-specific code is often implemented as assembly code to leverage low-level control. For example, context switching in an Operating Systems requires explicit control over the current processor state via architecture-specific assembly instructions. Porting such a sequence of instructions requires understanding the origin architecture and the precise semantics of each instruction.

We extend `Step 2` of our porting methodology with detailed semantic analysis of assembly code. By querying our KG, which contains hardware documentation, we can discover the effects an assembly instruction has on the machine state (e.g., memory access or processor state change). For any assembly routine, an LLM then extracts the combined semantic meaning of the sequence of instructions, which must be preserved during porting. Additionally, the LLM analyses the information in assembly labels and comments, which may hint at the original developer's intent. The collected resulting information is then stored in the KG for later steps in the porting workflow.

Preprocessor directives. While preprocessor directives are essential to enable portability of code, especially in C and assembly, the simple text replacement they provide can significantly alter the surrounding code [10]. For example, macros may change types, function attributes, or alter the control flow significantly. Typically, compilers require the user to specify all `define` macros (the configuration), perform the pre-processing step, and then analyze the resulting code. This necessarily omits analysis of other configurations, which may contain information that is useful or even necessary for the porting process. On the other hand, the number of configurations suffers from a combinatorial explosion, so checking all configurations one by one is generally infeasible. *Variability-aware parsing* is the process of syntactically analyzing source code while keeping track of *all* configuration options. This is conceptually similar to parsing all possible configurations individually, but attempts to share as much work as possible between parses, strongly improving worst-case complexity. Both TypeChef [19] and later SuperC [12] have shown that this way, parsing can still be performed in roughly linear time. Using these tools, we can track how system configuration and hardware-specific macro definitions affect the entire system, and include the resulting relations in our KG.

5 Outlook

Our future work will investigate the systematic evolution of the manual porting workflow presented in Section 4 toward an increasingly automated AAI-based framework. Building on the initial agent capabilities developed for `Step 2` and `Step 6`, subsequent research will extend agent support to the remaining stages of the workflow and establish mechanisms for agent orchestration, knowledge exchange, and verification. Particular attention will be given to integrating appropriate approaches, especially given the requirements of safety-critical development processes. In the long term, the objective is to enable the autonomous execution of increasingly complex development tasks.

The improvements achieved through static source code analysis will also need to be evaluated: While powerful LLMs may be able to understand large code bases purely with a naive RAG approach, we expect that supporting agents with static code analysis will reduce token usage and allow using smaller models. Future work will collect quantifiable metrics such as code quality and token usage to verify this.

A central research concern will be maintaining verifiability, traceability, and human control while increasing automation. This work thus aims to establish the foundations for a systematic, end-to-end agentic methodology for embedded systems engineering.

The current challenge in employing KGs for the porting use case is therefore the construction of a proper KG itself: one that ingests all necessary origin and target resources to ensure complete requirement traceability across the SRS and hardware specifications. This requires defining a proper knowledge representation, i.e., an ontology tailored to the porting domain, that specifies which entity and relation types are needed to capture this cross-domain traceability.

Acknowledgments

This work has been supported by the BMIMI, BMWET, and FFG, Contract No. 911655: "Pro²Future II"

References

[1] AUTOSAR. 2024. *AUTOSAR Classic Platform: Layered Software Architecture.* AUTOSAR. https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_EXP_LayeredSoftwareArchitecture.pdf Release R24-11. Accessed: 2026-08-10.

[2] AUTOSAR. 2026. AUTOSAR – AUTomotive Open System ARchitecture. https://www.autosar.org/. Accessed: 2026-09-01.

[3] Catalin-Virgil Briciu, Ioan Filip, and Franz Heininger. 2013. A New Trend in Automotive Software: AUTOSAR Concept. In *2013 8th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. 251–256.

[4] Sarfraz Brohi, Qurat-ul-ain Mastoi, N. Z. Jhanjhi, and Thulasyammal Ramiah Pillai. 2025. A Research Landscape of Agentic AI and Large Language Models: Applications, Challenges and Future Directions. 18, 8 (2025), 499. doi:10.3390/a18080499

[5] Manfred Broy. 2006. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*. ACM. doi:10.1145/1134285.1134292

[6] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, and Christian Salzmann. 2007. Engineering Automotive Software. *Proc. IEEE* (2007). doi:10.1109/JPROC.2006.888386

[7] Osvaldo De Souza and Helano S. Castro. 2007. A Complete Method for Porting Operating System for Embedded Systems. https://api.semanticscholar.org/CorpusID:208542455

[8] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. (2024).

[9] Lisa Ehrlinger and Wolfram Wöß. 2016. Towards a Definition of Knowledge Graphs. In *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems (SEMANTiCS) and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS) (CEUR Workshop Proceedings, Vol. 1695)*.

[10] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng.* 28, 12 (Dec. 2002), 1146–1170. doi:10.1109/TSE.2002.1158288

[11] Fortune Business Insights. 2026. Embedded Systems Market: Hardware & Software IT Services. Retrieved June 28, 2026 from https://www.fortunebusinessinsights.com/embedded-systems-market-108767

[12] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing all of C by taming the preprocessor. *ACM SIGPLAN Notices* 47, 6 (2012), 323–334.

[13] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. 1994. An Analysis of the Requirements Traceability Problem. In *Proceedings of the First International Conference on Requirements Engineering (ICRE)*. IEEE, 94–101.

[14] K. Grimm. 2003. Software technology in an automotive company - major challenges. In *25th International Conference on Software Engineering, 2003. Proceedings.* doi:10.1109/ICSE.2003.1201228

[15] Prasanna Hambarde, Rachit Varma, and Shivani Jha. 2014. The Survey of Real Time Operating System: RTOS. In *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*. doi:10.1109/ICESC.2014.15

[16] Kleber Nogueira Hodel, José Reinaldo Da Silva, Leopoldo Rideki Yoshioka, João Francisco Justo, and Max Mauro Dias Santos. 2022. FAT-AES: Systematic Methodology of Functional Testing for Automotive Embedded Software. *IEEE Access* (2022). doi:10.1109/ACCESS.2021.3128431

[17] HUAWEI Developers. 2026. ArkTS. https://developer.huawei.com/consumer/en/arkts/.

[18] IEEE. 1998. *IEEE Recommended Practice for Software Requirements Specifications*. Technical Report IEEE Std 830-1998. IEEE.

[19] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. Typechef: Toward type checking# ifdef variability in c. In *Proceedings of the 2nd international workshop on feature-oriented software development*. 25–32.

[20] Petri Kotiranta, Marko Junkkari, and Jyrki Nummenmaa. 2022. Performance of Graph and Relational Databases in Complex Queries. *Applied Sciences* 12, 13, Article 6490 (2022), 19 pages. doi:10.3390/app12136490

[21] Saravanan Krishnan, Alex Mathai, Amith Singhee, Atul Kumar, Shivali Agarwal, Keerthi Narayan Raghunath, and David Wenk. 2022. Incremental Analysis of Legacy Applications Using Knowledge Graphs for Application Modernization. In *Proc. Int'l Conf. on Software Engineering (companion volume)*.

[22] Vignesh Manjunath, Jesús Pestana Puerta, Tobias Scheipel, and Marcel Carsten Baunach. 2026. MiKB-RAG: A Microarchitecture Knowledge Base Framework for Embedded Software Development. In *Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing* (Thessaloniki, Greece) *(SAC '26)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3748522.3779941

[23] Renata Martins Gomes, Bernhard Aichernig, and Marcel Baunach. 2024. A framework for embedded software portability and verification: from formal models to low-level code. *Software and Systems Modeling* (2024). doi:10.1007/s10270-023-01144-y

[24] Ankur Maurya and Divya Kumar. 2020. Reliability of safety-critical systems: A state-of-the-art review. *Quality and Reliability Engineering International* (2020). doi:10.1002/qre.2715

[25] Congmin Min, Rhea Mathew, Joyce Pan, Sahil Bansal, Abbas Keshavarzi, and Amar Viswanathan Kannan. 2025. Towards Practical GraphRAG: Efficient Knowledge Graph Construction and Retrieval from Unstructured Text for Large-Scale RAG Systems. (2025).

[26] James D. Mooney. 1995. Portability and Reusability: Common Issues and Differences. In *Proceedings of the 1995 ACM 23rd annual conference on Computer science - CSC '95*. doi:10.1145/259526.259550

[27] James D. Mooney. 2001. *Issues in the Specification and Measurement of Software Portability*. Technical Report. West Virginia University.

[28] Robert Oshana. 2013. Software Engineering of Embedded and Real-Time Systems. In *Software Engineering for Embedded Systems*. Elsevier. doi:10.1016/B978-0-12-415917-4.00001-3

[29] Pebble Bay Consulting. 2025. Embedded Software Porting. Retrieved July 29, 2026 from https://pebblebay.com/embedded-software-porting/

[30] Sedat Sonko, Cosmas Dominic Daudu, Femi Osasona, Ayodeji Matthew Monebi, Emmanuel Augustine Etukudoh, and Akoh Atadoga. 2024. The evolution of embedded systems in automotive industry: A global review. *World Journal of Advanced Research and Reviews* (2024), 096–104.

[31] Annalisa Szymanski, Noah Ziems, Heather A. Eicher-Miller, Toby Jia-Jun Li, Meng Jiang, and Ronald A. Metoyer. 2025. Limitations of the LLM-as-a-Judge Approach for Evaluating LLM Outputs in Expert Knowledge Tasks. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*. doi:10.1145/3708359.3712091

[32] Susheng Wu, Ruisi Wang, Yiheng Cao, Bihuan Zhang, et al. 2025. Mystique: Automated Vulnerability Patch Porting with Semantic and Syntactic Enhanced Code Property Graphs. 2, FSE (2025), 130–152.

[33] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. doi:10.1109/SP.2014.44

[34] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Shengcheng Yu, Weisong Sun, Yun Yang, and Zhenyu Chen. 2026. A survey on large language models for software engineering. *Science China Information Sciences* (2026). doi:10.1007/s11432-025-4670-0

[35] Bo Zhou, Jiaqi Shi, Ying Wang, Li Li, Tsz On Li, Hai Yu, and Zhiliang Zhu. 2025. Porting Software Libraries to OpenHarmony: Transitioning from TypeScript or JavaScript to ArkTS. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025). doi:10.1145/3728941

[36] Müge Canpolat Şahin and Ayça Kolukisa Tarhan. 2025. A Multi-Perspective Review on Embedded Systems Quality: State of the Field, Challenges, and Research Directions. *Journal of Software: Evolution and Process* (2025). doi:10.1002/smr.70007