# Transforming Software Engineering and Software Acquisition with Large Language Models

John Robert and Ipek Ozkaya
Software Engineering Institute
Pittsburgh, PA, USA

Douglas C. Schmidt
William and Mary
Williamsburg, VA, USA

## ABSTRACT

Large Language Models (LLMs) are a form of generative AI that have become engine sof transformation in software engineering and software acquisition. Convenient access to chat-based LLMs trained extensively with vetted software data are enabling software engineers and developers to accelerate common tasks, including drafting requirements, generating code, and/or creating tests for common scenarios. Many development environments and tools now integrate with APIs offered by LLMs to create new interactive workflows.

There are pros and cons, however, of applying LLMs throughout the software development lifecycle (SDLC). One con is that LLMs can generate incorrect answers, such as code with known security vulnerabilities, semantically incorrect code even if it compiles, and software tests that are incomplete. One pro is LLMs can generate multiple alternative responses to a given request rapidly, allowing users to explore outcomes they otherwise may not be able to do. LLMs interact with users in the form of structured inputs, known as prompts, that can customize and improve LLM output. Prompts provide a powerful mechanism to develop and leverage patterns that improve the effectiveness of LLMs, including for common software engineering tasks. Effective prompt engineering can thus enhance the performance of LLMs for specific software development activities.

An open question addressed by this chapter is whether the transformative changes associated with LLMs yield more efficient software development methods and tools or do they simply create different software engineering workflows without meaningful improvements? To answer this question, we assess the impact of LLMs on common software engineering activities and combine these insights to discuss the impact of LLMs across the SDLC. Addressing this question is particularly important when considering the use of LLMs in high-stakes domains (such as healthcare, aerospace, and defense) where incorrect responses from LLMs can lead to catastrophic outcomes if not reviewed carefully by humans**.**

One goal of this chapter is to explore the challenges and opportunities for using LLMs to enable an AI-augmented software engineering future. Another goal is to evaluate LLM-based approaches that enable the detection—and ultimately the correction—of document incompleteness, inconsistencies, and discrepancies. Finally, we aim to demonstrate the transformative potential of LLMs in software engineering, focusing on how these models can enhance various phases of the SDLC when they are appropriately integrated with other effective automation approaches.

*Keywords: Large language models, LLM, generative AI, AI-augmented software engineering and software acquisition, software development lifecycle (SDLC), software assurance, detecting document inconsistencies and discrepancies.*

## I. INTRODUCTION

We depend on software for our daily lives. Software forms key parts of mobile, transportation, healthcare, entertainment, and national defense systems. Software engineering—the discipline of creating, deploying, and maintaining the software that we depend upon—has similarly become critical to building trust and assurance in software through engineering rigor.

Creating and maintaining mission- or safety-critical software is hard. Engineers must identify, architect, and build systems that meet quality attribute requirements, such as safety, predictability, or reliability. Tests must be performed to ensure systems meet their functional requirements and security standards. All these activities are performed as part of the software development lifecycle (SDLC) that guides software engineers in orchestrating iterative processes of software development, including bug fixes, new capability updates, and verification before software deployment. Many of these steps are repetitive, error prone, and time-consuming, which motivates software engineers to devise ways to automate key SDLC activities intelligently and reliably.

Artificial intelligence (AI) is both a construct of software engineering and a tool that can enable better software engineering. AI systems are built using software and data to create complex responses in many types of application domains, from product recommendation engines to self-driving cars. Applying AI concepts to the SDLC has been discussed for years (Barstow, 1987), with early examples applying machine learning (ML) to estimate software development around 1995 (Srinivasan & Fisher, 1995). The advent of powerful and accessible LLMs has the potential of reshaping SDLC (Nyuyen, S., 2024), with new opportunities and risks for software engineers (Ozkaya, 2023a).

This chapter examines if—and how—LLMs can be applied effectively throughout the SDLC. It also explores the potential benefits and challenges of LLMs in specific software engineering and software acquisition activities, such as development and testing, and evaluates their overall impact on the SDLC. We examine the impact of LLMs on multiple software activities in the SDLC, as well as the collective impact of LLMs across these activities via a case study for drone avionics that focuses on safety-critical software system development.

Traditional manual methods of verifying consistency between code and documentation are time consuming and do not scale up to larger software-reliant systems. Although LLMs are not consistency checkers, they can assist checking for patterns given a set of artifacts and enough examples in the training data. We therefore use our drone avionics case study to show how LLMs can assist with assessing compliance of code using mandatory regulatory standards codified in documents. To advance research on this topic, we introduce a novel framework that augments LLMs with advanced data structures to automate the detection of *document incompleteness*, *inconsistencies*, and *discrepancies* (DIID) in software related artifacts across the SDLC. DIID detection is useful for software developed for high-stakes domains to ensure relevant artifacts (e.g., requirements, architecture, design, code, and tests) conform to regulatory requirements (such as FDA, FAA, and DoD standards).

The remainder of this chapter is organized as follows: Section II discusses the expanding role of AI in the SDLC and shows how it is ushering in a new era of software engineering; Section III identifies key SDLC activities, opportunities, and risks for leveraging LLMs and summarizes recent research that examines the suitability of using LLMs in different SDLC activities; Section IV explores techniques for detection of DIID and summarizes the drone avionics safety-critical case study to showcase LLM-based DIID detection challenges and opportunities; and Section V presents concluding remarks and outlines future research and advancements, including architectural and testing considerations.

## II. Exploring the Expanding Role of AI in the Software Development Lifecycle (SDLC)

The impact of AI on the SDLC has multiple dimensions. The discipline of software engineering has experienced several transformative changes in recent decades, ranging from the adoption of higher-order functional programming languages to advances in automation using DevSecOps tools and processes. A consistent theme during these times of transformation is that it's often hard to foresee how far changes will go as the hope (and hype) meets the realities of the technical and programmatic limitations.

AI is influencing software engineering both in terms of replacing or augmenting capabilities with AI-augmented approaches and increasing intelligent automation to support software engineering activities. Figure 1 expands upon a vision presented in the book *Architecting the Future of Software Engineering: A National Agenda for Software Engineering Research & Development (Carleton et al., 2021).* This figure depicts key dimensions of delivering AI capabilities and intelligently automating the SDLC by applying AI augmentation in both system operations and software engineering activities, ranging from conventional to fully AI-augmented methods (Robert et al., 2024).

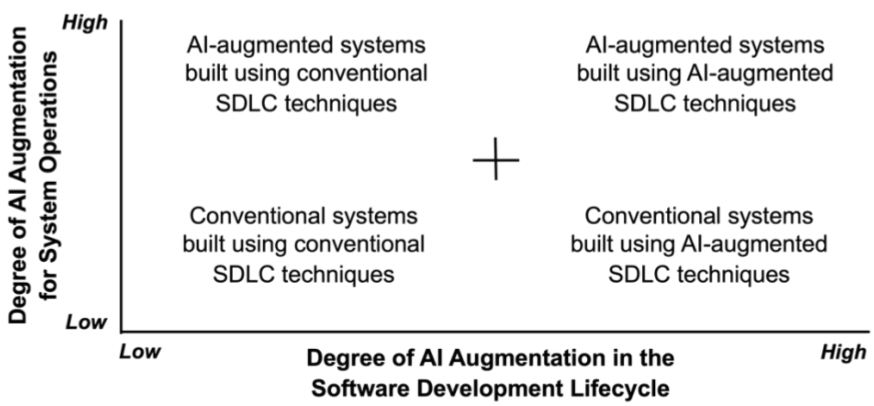


***Figure 1: Taxonomy of AI Augmentation for System Operations & Software Development Lifecycle Activities.***

Each portion of Figure 1 is summarized below, starting with the lower-left quadrant.

- **Conventional systems built using conventional SDLC techniques**. This quadrant represents a low degree of AI augmentation for both system operations and the SDLC, which is the baseline for most software-reliant projects heretofore. An example is an avionics mission computing system that operates via distributed object computing middleware and rate monotonic scheduling (Harrison et al., 1997) and is developed using conventional SDLC processes without any AI-augmented tools or methods.
- **Conventional systems built using AI-augmented techniques**. This quadrant represents an emerging area of research, development, and practice in the software engineering community, where system operations have a low degree of AI augmentation, but AI-augmented tools and methods are used in the SDLC. An example is a website hosting service where the content is not augmented by AI, but the development process employs AI-augmented code generators (such as GitHub Copilot), code review tools (such as Codiga), and/or testing tools (such as DiffBlue Cover).

- **AI-augmented systems built using conventional SDLC techniques.** This quadrant represents a high degree of AI augmentation in systems, especially in their runtime operations, but uses conventional methods in the SDLC. An example is a recommendation engine in an e-commerce platform that employs machine learning algorithms to personalize recommendations, but the software itself is developed, tested, and deployed using conventional Agile methods and the React.js and Node.js frameworks.
- **AI-augmented systems built using AI-augmented techniques**. This quadrant represents the pinnacle of AI augmentation, with a high degree of AI-augmentation for both systems operations and the SDLC. An example is a self-driving automotive system that operates via ML algorithms for navigation and decision making while also using AI-driven code generators, code review and repair tools, unit test generation, and DevOps tools for software development, testing, and deployment.

Although the majority of SDLC activities today are performed in the lower-left quadrant (*i.e.*, conventional systems built using conventional SDLC techniques), the trend and innovation trajectory is towards the upper-right quadrant, where an AI-augmented SDLC is used to build and deploy AI-augmented operational capability. This trend has accelerated over the last decade starting with increased reliance on AI-augmented capabilities as part of systems and recently with more AI-augmented tools coming to market and being applied to develop, test, and deploy software. In that context, however, a range of new challenges have emerged, such as the fast pace of technology changes, managing modifications to software development workflows, and understanding potential bias and mistakes in the training corpus of AI-augmented tools (Panyam, 2024).

Integrating LLMs into the SDLC requires a measured approach, balancing concerns like disclosure, accuracy, and ethical use (Robert & Schmidt 2023). Success hinges on developing organizational policies for these concerns and adapting to evolving governance and regulations. An empirical understanding of workflow alterations and data collection helps inform decisions about the success of new approaches. Moreover, traditional practices, such as code reviews with customized checklists, may even regain prominence, providing humans in the loop with tools and methods needed to accelerate the reliability and testability of code and systems developed by leveraging the assistance of LLMs.

## III. APPLYING LLMS THROUGHOUT COMMON SDLC PHASES

A recent literature survey (Hou et al, 2024) on applying LLMs to software engineering reviewed 395 research papers from January 2017 to January 2024 categorizing where the applications of research is in applying LLMs in software engineering. Their findings indicate that 62% of the investigations focus on software implementation related activities. Given our summary of opportunities of applications of LLMs to different activities across the SDLC, studying the limits of how to leverage LLMs is an ongoing research area with implications in tool and method development and application.

We need to assess the following when considering whether LLMs can assist with software engineering tasks to enhance SDLC processes that improve system capabilities:

1. *How LLMs can improve a single task*, focusing largely on doing something existing tools and engineers can do better, *e.g.*, code review or code completion,
2. *How applying LLMs on single or integrated task workflows can help consistency* with respect to other artifacts and activities related to that task and its inputs/outputs, *e.g.*, propagating a refactoring change consistently to the rest of a system or ensuring consistency of artifacts across multiple artifacts, and
3. *How applying LLMs may result in workflow shifts resulting in further improvements* by understanding the dependencies of a single task on related artifacts and task workflows, e.g., enabling different orchestrations of activities and a new SDLC augmented by LLMs.

As research and data on the outcomes of these applications continue to grow, we expect they will influence the development of new SDLC processes (Ozkaya, 2023a).

This chapter focuses largely on the impact of LLMs on single SDLC tasks, discusses implications on the dependencies of these tasks to other activities, and describes how LLMs can be applied to enable orchestration across multiple SDLC activities. Although applying LLMs in the SDLC involves AI performing some work traditionally done by people, many basic principles and practices remain relevant. In particular, many tools, techniques, and procedures used to ensure confidence in the validity and verification of software still apply, though LLMs and associated generative AI tools will now perform more of the workload.

The remainder of this section summarizes recent research that examines the application of LLMs in different software engineering tasks mapped to the SDLC phases shown in Figure 2 below. Understanding the limits of LLMs applied to software engineering tasks is a fast moving area of model development, tool development, and research. This section thus provides examples of ongoing work and is not intended as a comprehensive survey of the state of research on LLMs to date.

*A. Assessing the Opportunities for Applying LLMs in SDLC Phases*

Contemporary SDLC models are iterative, reflecting a continuous cycle of software professionals utilizing tools that enable greater automation. Figure 2 visualizes a high-level generalization of some common activities in SDLC phases, focusing on where LLMs can be—and are being—applied at each phase. While the orchestration of these activities may vary depending on what SDLC model a team uses, all software projects execute these activities in some order to deliver software.

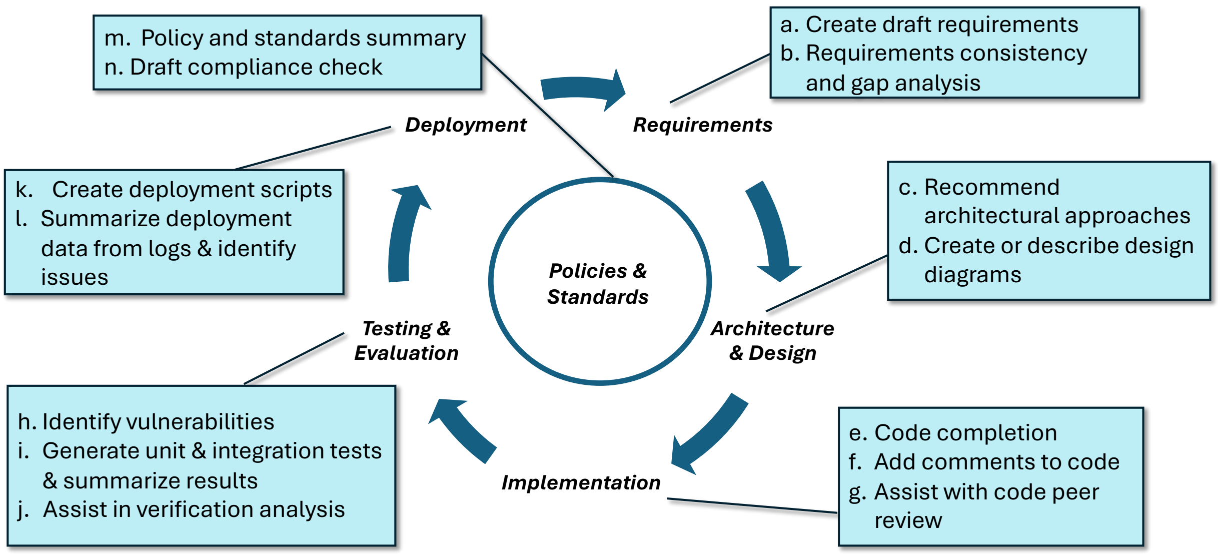


***Figure 2: Opportunities to Apply LLMs to Software Development Lifecycle (SDLC) Activities.***

We next describe each SDLC phase and summarize research on applying LLMs to improve specific activities by accelerating the efficiency of generation while not compromising from quality. An implicit assumption in applying LLMs to any activity, including software engineering activities, is that the LLM has been trained with the relevant data enabling it to produce relevant outcomes. The relevance of the training corpus of an LLM will vary from domain-to-domain and activity-to-activity.

For example, systems that have been the subject of public discussion and debate (*e.g.*, patient enrollment websites associated with the Affordable Care Act in the USA) may have more to draw from for LLM requirements. Conversely, activities that are not consistently represented by artifacts in the public domain may incur lower rates of LLM accuracy. As a case in point, while there are many good open-source software repositories, there are few examples of good open-source software architecture documents to train LLMs upon.

*1) Requirements*

Activities in the *Requirements* phase involve collecting, specifying, and understanding the needs, use cases, and constraints of a software-reliant system. During this phase, specifications are created that define what the software should do, including functional requirements (*e.g.*, features and functionality) and quality attribute requirements (*e.g.*, performance, security, and usability). These requirements can be documented in many formats, such as documents listing specifications, use cases describing expectations of the system, compliance and policy documents providing constraints, and/or diagrams describing expected interactions with users. This phase includes a significant amount of discovery and interaction with the key stakeholders and the analysis needed to ensure that requirements are clear, complete, and supported by stakeholders. These requirements are then prioritized with stakeholder feedback to ensure the most valued requirements are implemented and improved in subsequent development cycles.

Ongoing research investigates using LLMs to (1) elicit requirements, including examining requirements documents, feature requests, and user feedback, and (2) identify themes or areas of high importance. For example, an area of ongoing research elicits requirements by generating initial drafts of requirements based on project descriptions or stakeholder inputs (Abbas, 2023). Recent research (Tikayat Ray et al., 2023) has found that LLMs can accelerate requirements by converting requirements to machine readable formats. Prompt engineering patterns have been identified to accelerate elicitation of requirements (White, Hays, et al., 2023) and assist in prioritization of requirements (Sami et al., 2024). LLMs can also be used to help identify contradictions in software specifications (Gärtner & Göhlich, 2024).

A detailed document called the *Software Requirement Specification* (SRS) is typically output from the *Requirements* phase, especially in regulated environments. Other outputs include iteratively identified use cases, formal specifications, and architecturally significant requirement documents. There are multiple elicitation and documentation activities that are expressed using natural language, which make LLMs well-suited to assist software engineers during the *Requirements* phase. As shown in Figure 2, some examples include the following:

a. **Create draft requirements or use cases** – LLMs can rapidly convert short text summaries of ideas or requests into more readable and requirement descriptions. Prompt engineering can provide additional structure and guidance to create draft requirements statements that are relevant for the software domain and also structured in a desired format. LLMs can also generate alternative use cases that describe how software will interact with users or other systems. It is important to highlight that these will only be draft requirements because human software engineers who understand the software domain, standards, and risks should review, edit, and approve the final requirements. However, using LLMs to create draft requirements quickly—and in some cases with alternatives—can accelerate the requirements elicitation process, provide examples of edge scenarios, and accelerate the pace of iteration, resulting in improved draft documents during the *Requirements* phase. Moreover, LLMs can automatically generate use cases, scenarios, and requirement specifications from natural language inputs.

b. **Requirements consistency and gap analysis** – LLMs can assist in interpreting and analyzing stakeholder inputs, analyzing an SRS for completeness or consistency, and/or convert these stakeholder inputs into formatted requirements documents through summarization. LLMs can help users identify common concepts across requirements or categorize requirements. They can also suggest requirements that may be missing or are incomplete by processing vast amounts of data, including previous project requirements, industry standards, policy documents, and best practices.

Leveraging LLMs throughout the *Requirements* phase can enable increasing the pace of idea generation and requirements elicitation. This approach, however, requires teams to understand their baseline and goals, so that they can properly assess the quality of LLM outputs.

The *Requirements* phase informs the *Architecture and Design* phase since a system should be architected and designed to meet the requirements it must achieve.

### *2) Architecture and Design*

The *Architecture and Design* phase defines and describes the structure, components, and interfaces of a system's software. This phase includes high-level architectural planning, architectural analysis, prototyping to compare trade-offs or assess response measures, and detailed design specifications. During this phase, engineers make architectural decisions to satisfy quality attribute requirements (such as performance and security) and perform experimentation or analysis to verify that the architecture and design perform as intended. These architectural decisions are captured in architecture design records, architectural diagrams, design diagrams, prototypes and accompanying text that describes the architectural elements and interfaces.

The output of the *Architecture and Design* phase is a set of architecture decisions, diagrams detailing these decisions, and the descriptions of key constraints, requirements, and trade-offs that provide the context for the decisions. These artifacts are typically captured in a *Software Architecture Document* (SAD) that provide an overall description of the software architecture and each architecture element than needs further refinement, including its responsibilities and interfaces. Some teams apply lightweight documentation approaches and capture these details in diagrams or *architecture design records* (ADRs). Large organizations and highly-regulated projects may need to also to develop a *Detailed Design Document* (DDD) that describes each component, as well as an *Interface Control Document* (ICD) that provides data exchange details for all interfaces.

Applying LLMs to support software architecture and design has been sparse and mostly exploratory investigations to date. The area seeing most attention is architecture knowledge generation (Ozkaya 2023b). An exploratory empirical study shows that while LLMs can document and formulate design decisions in the structure of ADRS for a given context, their quality is not commensurate to those generated by architects, though fine tuning can improve their results (Dhar et al., 2024). Likewise, experiments with high-level architecture-relevant prompts show that software architects value the quick alternative generation ability of LLMs, but also consider trickier issues, such as copyright concerns and hallucinations, as critical barriers to adoption (Jahic and Sami, 2024).

There are also ongoing investigations to generate code from given images in the domain of control software. For example, (Koziolek et al 2024) show how to generate IEC 61131-3 Structure Text control logic source code from piping-and-instrumentation diagrams with LLM-trained image recognition. As these capabilities mature, generating code from image recognition use cases can be generalized to other situations where code is generated from architecture diagrams, provided these diagrams have sufficient detail and formal semantics that an LLM can map to patterns of implementation details.

As shown in Figure 2, examples to leverage LLMs in the *Architecture and Design* phase including the following:

**c. Recommend architectural approaches:** A strength of LLMs is their access to general knowledge. A significant amount of information across domains and systems exists about common architecturally-significant requirements, such as performance, security, availability, and scalability, and experiences in implementing these requirements also exists. Using LLMs to brainstorm architectural design alternatives to given problems can thus improve a team's ability to analyze their designs and supplement their knowledge with existing patterns and tactical information. This analysis can take the form of providing sufficient context to an

LLM and asking it to generate output relevant for similar systems, patterns, tactics, and technologies. Architects can use this LLM output to explore design spaces during the *Architecture and Design* phase.

d. **Create or describe design diagrams** – LLMs can assist generating diagrams to assist architecture documentation during the *Architecture and Design* phase. For example, LLMs can take high-level descriptions of system components, their relationships, and interactions as input and automatically generate visual representations, such as UML diagrams, flowcharts, and data flow diagrams. By interpreting the textual descriptions of the architecture and translating them into structured visual outputs, LLMs can facilitate refining existing diagrams by suggesting improvements based on patterns they learned from vast amounts of architecturally-relevant training data. This capability can enable teams to quickly prototype and iterate on designs. When diagrams are involved as part of the tasks, the underlying assumption is that an LLM is multi-modal, *i.e.*, it can work with images, video, and/or audio, in addition to text.

   LLMs can also provide detailed descriptions and explanations of design diagrams created during the *Architecture and Design* phase. By analyzing the structural elements and relationships depicted in diagrams, LLMs can generate initial narratives that elucidate the purpose, functionality, and interactions of various components within a system. These descriptions can explain a system's modules, data flows, and interface connections, as well as how the design addresses specific requirements, such as scalability, security, and performance. Moreover, LLMs can identify potential patterns reflected in the diagrams, offering insight that can aid understanding and refining the architecture. This capability makes LLMs valuable tools for documenting, summarizing, and communicating design concepts, assisting stakeholders establishing an understanding of system software architectures.

The designs generated during the *Architecture and Design* phase should be reified during the *Implementation* phase to build the actual software, thereby ensuring it meets functional and quality attribute requirements, such as safety, security, reliability, and reusability.

### *3) Implementation*

The *Implementation* phase involves coding and developing the software components according to requirements and within the constraints identified during the *Architecture and Design* phase. Programming is performed in this phase to create new software capability, fix previously identified bugs, and/or perform other modifications. This phase includes not just writing code, but also commenting the code, integrating the new code into revision control systems and DevOps pipelines, and conducting software quality control activities, such as software peer reviews and static/dynamic tool analysis.

Software developers are increasingly using LLMs to generate code, usually in response to prompts in a browser window. Moreover, coding assistants, such as GitHub Copilot and Amazon CodeWhisperer, are being merged with popular integrated development environments, such as IntelliJ, Android Studio, Visual Studio, and Eclipse. In both cases, creating code from prompts can increase developer productivity. Moreover, these AI code assistants include other capabilities, such as code refactoring to modify existing code. There is also ongoing work focused on investigating with code transformations, translating code it into different programming languages, programming language versions, and/or computing platforms (Pan et al., 2024).

As recent systematic literature reviews have demonstrated, code generation is where LLMs often excel, especially for popular programming languages like Java and Python. For example, LLMs have demonstrated effectiveness at auto code completion (Bird et al., 2023). LLMs can help developers learn new programming languages or frameworks by providing instant code examples and explanations (Lones, 2024). Likewise, LLMs enhance code documentation by generating summaries and explanations, making it easier for developers to understand and maintain codebases. A recent example (Geng et al., 2024) highlights the benefits in code comment generation, which is a common use case for software engineers. LLMs also enable easily and rapidly creating sample code or prototypes that contribute to learning and applying software concepts rapidly (Jiang et al., 2022). Other ongoing research includes creating reusable libraries for specific domains (Grand et al., 2024), supporting code reviews (Li et al., 2022) and refactoring code when integrated with static analysis tools (Pomian, et al. 2024).

Common outputs from the *Implementation* phase include new source code files, code comments, and code commit history into version control systems as well as code peer review or other quality control summaries. As shown in Figure 2, there are multiple opportunities to leverage LLMs in the *Implementation* phase, including the following:

**e. Code completion:** Code completion is an activity that software engineers are utilizing LLMs for effectively to date. This activity leverages LLM strengths in pattern matching to suggest recommended ways for completing implementations as developers use IDEs. In such activities the context scope is narrow and relevant data are available, which minimizes the likelihood of LLM hallucinations and other implementation mistakes.

f. **Adding comments to code** – LLMs can be used to add comments to software written by programmers, enhancing code readability and maintainability. By analyzing the structure and functionality of the code, LLMs can automatically generate meaningful comments that describe the purpose of methods and classes, the role of variables, and the logic behind specific code blocks. These comments can be tailored to various levels of detail, from high-level summaries of entire modules to line-by-line explanations of complex algorithms. Moreover, LLMs can be integrated into IDEs and development workflows to ensure comments remain up-to-date as code evolves by suggesting updates to comments after modifications. This automation reduces the burden on developers to document their code manually, enabling them to focus on writing efficient, high-quality software while ensuring the codebase remains well-documented for future developers and maintainers.

g. **Assist with code review and bug fixing** – LLMs can be applied to review code as a peer review assistant (Li et al., 2022). Recent LLMs have been trained extensively on valid code and can assist with bug fixing (Xia et al., 2023). LLMs can also summarize the results of multiple peer reviews (including those by humans) to identify trends or common issues from reviews. Likewise, LLMs can compare code commits to planned code additions or modifications to identify code changes.

The *Implementation* phase is followed by *Testing and Evaluation* activities, where the developed code is compared and assessed against relevant requirement and design specifications.

*4) Testing and Evaluation*

The *Testing and Evaluation* phase focuses on conducting activities to ensure (1) software meets its specified requirements and (2) implementations adhere to commonly agreed upon standards to avoid mistakes resulting in unintended bugs and security issues. This phase includes multiple testing levels, such as unit, integration, system, and acceptance testing. Thorough testing is crucial for evolution and sustainment since it validates that the software is reliable and ready for deployment and ongoing maintenance. Testing commonly includes functional and non-functional aspects of the system, as well as user testing.

The ability of LLMs to convert natural language prompts into software testing activities reduces the manual effort required for these important tasks (J. Wang et al., 2023). For example, LLMs can also accelerate test automation and development of testing sequences (Z. Liu et al., 2024).

Outputs of the *Testing and Evaluation* activities typically include test plans, test cases or scenarios, and test reports that summarize all testing activities. Opportunities to leverage LLMs in the *Testing and Evaluation* phase have similar structure with some activities discussed in the *Requirements* and *Implementation* phases, ranging from document generation and analysis to code generation in the form of unit tests. As shown in Figure 2, some examples include the following:

h. **Identify vulnerabilities** – LLMs can review code and help humans identify possible vulnerabilities or insecure programming practices. Secure coding practices are documented (*SEI CERT Coding Standards*) and always evolving, making it hard for humans to keep abreast of the latest practices. When given examples, LLMs can examine code to identify possible vulnerabilities and in multiple programming languages (Purba, M.D.). Early LLMs, such as ChatGPT-3.5, demonstrated mixed results to detect vulnerabilities (*Using ChatGPT to Analyze Your Code?*), but results continue to improve with good prompt engineering (Liu, Z., Yang, Z., Liao, Q. 2024) and with the latest improvement in LLMs (Sherman, M. 2024).

i. **Generating unit and integration tests and summarize results** – Applying LLMs to create test cases is a rapidly evolving focus area. These tools can enable interactions with software engineers and analysts to explore code interactively in ways relevant to testing activities, including asking for code summaries, checking compliance with coding standard(s), and/or exploring how code relates to specific considerations, such as safety, security, or performance. Generation of unit tests is an activity with a known structure and limited context scope, which is a good fit for LLMs.

j. **Assist with verification analysis** – LLMs can help identify inconsistencies in verification analysis during the *Testing and Evaluation* phase, leveraging their strengths in document processing and summarization. Software engineers can accelerate their analysis of test cases, test results, and the corresponding requirements and design specifications by using summarization prompts. For example, LLMs can compare between expected and actual test outcomes, potentially alerting these discrepancies. Likewise, LLMs can analyze text in documentation and test reports to identify patterns that do not match, potentially detecting conflicting interpretations or ambiguities that might yield inconsistencies. After these gaps are identified, developers can also utilize LLMs to suggest possible corrective actions or refinements to both the code and test cases. Section IV explores these capabilities in more detail and provides a case study example.

The *Testing and Evaluation* phase is followed by the *Deployment* phase, where tested code is released into operational environments and systems. DevSecOps tools and techniques connect SDLC testing and deployment activities into incremental and automated iterations that enable rapid, reliable, and repeatable releases of new software.

*5) Deployment*

The *Deployment* phase involves releasing software into operational systems. This phase also collects data related to operational effectiveness to inform maintenance and enhancements post-deployment, thus ensuring systems remain aligned with user needs and adapt to changing requirements. Moreover, the *Deployment* phase identifies bug fixes, updates, and new features to provide feedback from this phase to other phases for future software releases.

LLMs can play an important role in supporting software engineers during the *Deployment* phase, which is an area being increasingly automated. One example is program repair (Jin et al., 2023), which fixes issues found as deployment scripts are run to perform software updates. DevSecOps pipelines are used extensively across software development operation and management, and LLMs provide new opportunities to automate and scale software pipeline activities (*How to Put Generative AI to Work in Your DevSecOps Environment*, 2024). Early examples (Gurunathan, 2024) integrate LLMs into software repositories and include creating YAML scripts.

Artifacts created during the *Deployment* phase typically include deployment plans (such as rollback procedures, updated deployment scripts, and configuration files), and release notes documenting changes to software, as well as minimal requirements for hardware and software environments. As shown in Figure 2, multiple activities can leverage LLMs in the *Deployment* phase, including the following:

k. **Create deployment scripts** – This activity targets generating code relevant to deployment automation, which is another example of code generation where LLMs excel. Given specific requirements of a deployment environment as input prompts, LLMs can generate relevant automation scripts that support setting up servers, configuring environments, and deploying software components. These generated scripts handle a wide range of tasks, including database migrations, environment variable configurations, load balancing, and rollback procedures when deployments fail, all of which have existing patterns and exemplar reusable code that enable LLMs to generate robust code. When guided with structured prompts, LLMs can also adapt these scripts to different deployment platforms, such as cloud services, on-premises servers, or containerized environments, ensuring the deployment process is smooth and consistent across heterogeneous infrastructure. Similar to other code generation tasks, software engineers can apply LLMs to guide them through processes that optimize deployment scripts by identifying potential inefficiencies and suggesting improvements. For example, LLMs can reduce the time and effort to write scripts manually and minimize the risk of errors during deployment.

l. **Summarize deployment data from logs and identify issues** – By processing large volumes of log data generated during deployment, LLMs can extract key information, such as successful operations, warnings, and errors. They can then generate concise summaries that highlight important events and patterns, making it easier for developers and operations teams to understand the overall status of a deployment. Likewise, LLMs can analyze log data to help detect anomalies, potential bottlenecks, or recurring issues that might indicate underlying problems. Employing LLMs in these summarization tasks can help humans examine large volumes of data and possibly identify problems more rapidly than conventional manual reviews, thereby enabling teams to take corrective actions swiftly and ensuring a smoother deployment process.

### *6) Policies and Standards*

*Policies and Standards* have implications across all SDLC phases, as shown in Figure 2 (which places these artifacts at the center of the SDLC). These artifacts include software standards, such as coding or architectural standards, as well as standards that set expectations for quality attributes, such as safety, security, and reliability. Software must also adhere to policies established by organizations that develop software and may also include policies set by government regulators or acquisition programs. Compliance with software standards and policies is important for each phase of the SDLC, as well as for each iteration of software releases.

The powerful features provided by LLMs motivate a focus on Generative AI capability models that characterize levels of functionality provided by LLMs to assist organizations adopt to using Generative AI services intentionally and by avoiding their pitfalls. Much like software capability maturity models were used to assess the ability of organizations to create robust software solutions, LLM capability models are being proposed to help organizations measure their ability to create or deploy LLMs in their business processes effectively. Several capability models have been suggested, including models that describe how to leverage LLMs for applications (*Generative AI Capability Model*, n.d.), as well as models that explore the use of generative AI more broadly across organizations (*The MITRE AI Maturity Model and Organizational Assessment Tool Guide*, 2023).

Figure 2 shows the following ways that LLMs can be leveraged to create *Policies and Standards*:

m. **Policy and standards summary** – LLMs can be used to analyze policy and standards documents and provide summaries for humans (Bright et al., 2024; Robert & Schmidt, 2024). Many software engineering activities require conforming to governance and regulatory documentation across the SDLC. For example, software engineers perform many other tasks beyond coding, such as participating in regulatory compliance meetings, examining regulatory documents, or interacting with different industry or government standards stakeholders. These activities historically require humans to inspect and summarize reams of documentation manually. LLMs can help humans perform those activities more efficiently and accurately, as well as helping improve the quality and efficiency of humans involved with government software acquisition activities and policies. LLMs can also be used for document summarization and decision support, though workflows must ensure that human judgement is central to—and designed into—these processes due to limitations with conventional LLMs.

n. **Draft compliance check** – The pattern matching capabilities of LLMs enable them to process and understand large volumes of textual data. These capabilities can be leveraged to analyze software documentation, code, and design artifacts automatically and then ensure they adhere to established organizational policies and industry standards. These LLMs can compare the content of such artifacts against predefined requirements, such as coding standards, security protocols, or regulatory guidelines, to identify discrepancies or non-compliant elements. Using LLMs to draft compliance checks following given templates as patterns will assist end users focus on other areas, potentially reducing the risk of overlooking compliance issues, assisting necessary quality and regulatory expectations throughout its development lifecycle are captured.

The example LLM activities shown in Figure 2 and described above are relevant to current and emerging software engineering use cases. However, leveraging the knowledge of experienced software engineers remains vital to avoid overreliance on generative AI tools that have not reached high levels of maturity and trust. What is new is the interactivity provided by LLMs that enables software engineers to explore answers to questions and iteratively develop solutions to common SDLC problems.

*B. Assessing the Risks for Applying LLMs in SDLC Phases*

Determining whether or not to apply LLM capabilities throughout the SDLC requires a framework for assessing the risks. One framework of assessing the risks of using an LLM in the SDLC appears in (Bellomo et al., 2023). Figure 3 visualizes that risk assessment framework using two dimensions: (1) the consequences of mistakes made by an LLM (from low to high on the vertical axis) and (2) the time and effort needed to detect LLM mistakes (from high to low on the horizontal axis). The application domains

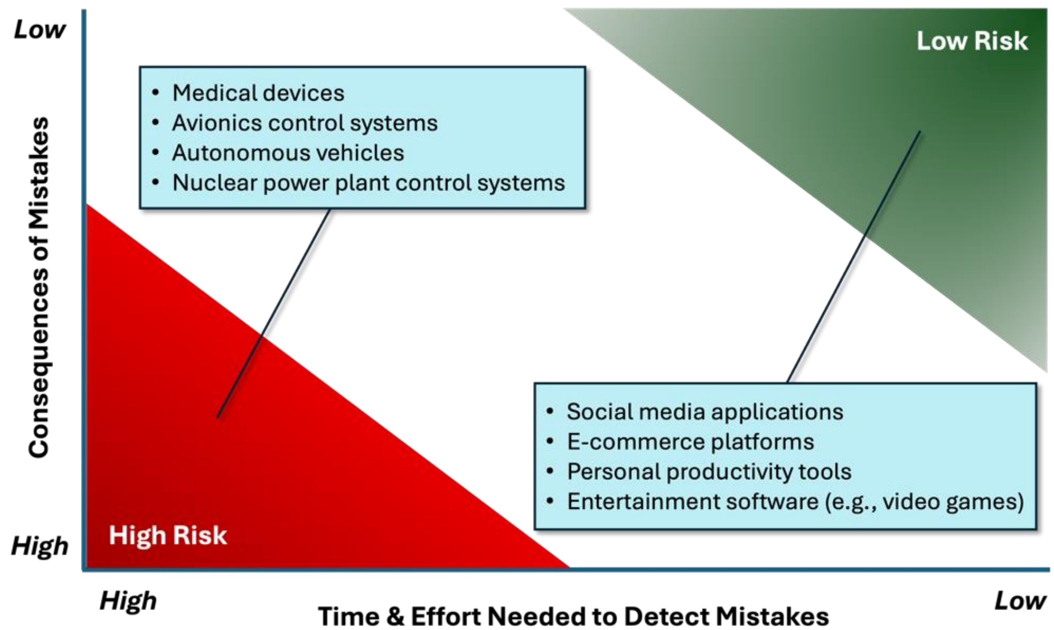


***Figure 3: Evaluating Risks Associated with Applying LLMs to Various Types of Software Systems.***

(such as medical devices, avionics control systems, autonomous vehicles, and nuclear power plant control systems) listed in the lower left-hand corner of the figure are classified as "high risk" because errors or failures in these systems can lead to catastrophic consequences, including loss of life, severe environmental damage, or large-scale safety hazards. These domains involve critical safety functions where precision, reliability, and security are paramount. Mistakes made by an LLM in these contexts could go undetected due to system complexity, making them particularly dangerous. This high risk is exacerbated since detecting and correcting errors in these environments often requires significant time and effort given the intricacy of the systems, the potential for widespread impact, and additional compliance checks triggered during system development and/or operation.

In contrast, the application domains (such as social media applications, e-commerce platforms, personal productivity tools, and entertainment software) in the upper right-hand corner of the figure are considered "low risk" because mistakes made by LLMs in these domains typically have less significant consequences. For example, errors in these domains might lead to user inconvenience or financial loss but are unlikely to harm individuals or environments. Moreover, mistakes in these applications are generally easier to detect and correct due to their lower complexity, as well as the ability to iterate and update relevant software rapidly. As a result, the time and effort required to identify and mitigate errors in these lower-risk domains is relatively small, making them more tolerant of mistakes generated by LLMs.

Given the assessment of risks described above, it is important to reiterate that humans are an essential part of using generative AI tools as part of any process and should not be replaced wholesale at this stage of their maturity. Moreover, given the nascent nature of the first-generation of LLMs applied in software engineering, it's essential to have skilled software and systems engineers, as well as subject matter experts, who can spot where generated documentation or code is inaccurate and ensure that the key context is not lost. These human skills are important and necessary, even as generative AI tools continue to improve and provide significant new capabilities.

For example, today's LLMs that generate code have been trained on imperfect code from open-source repositories, such as GitHub and Stack Overflow. Not surprisingly, the code they generate may also be imperfect (*e.g.*, there may be defects and vulnerabilities). It's therefore essential to leverage human insight and oversight across the SDLC through all the *Requirements*, *Architecture and Design*, *Implementation*, *Testing and Evaluation*, and *Deployment* phases shown in Figure 2.

*C. AI Augmentation Across SDLC Phases*

The application of AI across SDLC phases can also be expressed in terms of how much of the SDLC is supported with LLMs. As mentioned in earlier sections, LLMs can be applied to specific SDLC phases, but what if an LLM, or combination of LLM agents, assist humans across *multiple* SDLC phases in a coordinated or integrated approach? The application of LLMs to the SDLC can be viewed at different layers of the SDLC, each with specific benefits. **Table 1** below provides a summary of LLM application from this multi-layered SDLC perspective.

***Table 1: Layers of LLM Application to SDLC***

| SDLC LAYER | DESCRIPTION |
|---|---|
| None | No use of LLMs in the SDLC |
| Localized | Application of LLM to specific SDLC phase or phases, without coordination across phases |
| Cross-functional | Application of LLMs across multiple SDLC phases, with coordination across multiple phases to assist humans across multiple SDLC phases |
| Orchestration | Application of LLMs across all phases of the SDLC to assist humans in all aspects of the SDLC |

This multi-layered SDLC view provides additional insights into how LLMs can be applied to the SDLC. For example, consider a hypothetical SDLC where LLMs are applied as described in the previous section in localized phases, including *Implementation*, *Testing and Evaluation*, and *Deployment*. Connecting these three SDLC phases with another LLM to assist humans to optimize across these activities is an example of cross-functional layer, as described in **Table 1**. Extending this concept, an LLM that assists humans across all phases of the SDLC is an example of orchestration, which enables broader insight and assistance to human software developers. Linearizing the iterative SDLC depicted in Figure 2, and adding the layers from *Table 1*, the application of LLMs across the SDLC is visualized via the layers shown in **Figure 4**.[1]

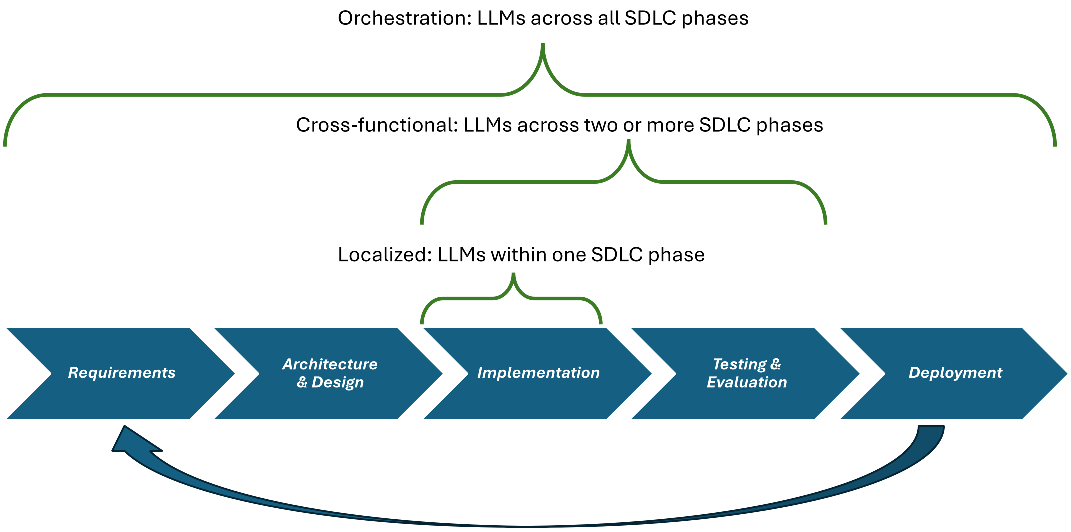


***Figure 4: Layers of LLM Application to SDLC***

In addition, this multi-layered view guides what SDLC data would be need to enable use of LLMs across the SDLC. For example, an LLM that connects multiple SLDC phases, such an *Implementation* and *Deployment*, must have access to data across these SDC phases. This dependency may require LLMs to understand different data types, such as code and testing results. It may also require

[1] The arrow from *Deployment* to *Requirements* reflects the iterative nature of these activities, repeated through the SDLC.

prompt engineering from human developers on the most important SDLC aspects, such as minimizing software defects for safety-critical systems or performing fuzz testing to augment security analysis. These higher level perspectives point to a future state with flexible and highly interactive human/LLM teaming that enables humans to prioritize the most critical tasks across the SDLC.

## IV. Detecting Incompleteness, Inconsistencies, & Discrepancies in Standards and Policy Documents

SDLC activities must often adhere to national and/or international regulatory requirements to ensure security, safety, and other important properties in high-stakes domains, such as healthcare, automotive, and aerospace. For example, all medical device software in the United States must follow Food and Drug Administration (FDA) software standards. Likewise, software in cars and trucks must follow regulatory standards established by the National Highway Transportation and Safety Agency (NHTSA). Moreover, software in airplanes and drones must comply with standards set by the Federal Aviation Administration (FAA). These software standards are described across multiple (often large) documents, ranging from directives or standards that require compliance to more general guidance that recommends best practices or common approaches.

Completeness and consistency between regulatory documents and project-specific software documents or artifacts should be confirmed by software developers and regulatory authorities and re-verified throughout the SDLC in high-stakes domains. This section explores techniques for detecting *document incompleteness*, *inconsistencies*, and *discrepancies* (DIID). In this section, we explore whether generative AI technologies in general—and LLMs in particular—offer new opportunities to automate DIID detection.

Section V.A defines what is meant by DIID detection in regulatory documents, identifies research gaps, and proposes a testing framework to quantify strengths and weaknesses in different architectural solutions. Formal definitions for incompleteness, inconsistency, and discrepancy are provided in Section V.B to help identify and summarize relevant related research. A representative software engineering case study is presented in Section V.C to demonstrate traceability from the generalized software engineering policies, standards, and guidance documents to DoD system-specific software documents and information. Related worked on DIID detection is evaluated in Section V.D.

### A. *Motivating the Need for Automated DIID Detection*

Incompleteness, inconsistencies, and/or discrepancies between regulatory documents and project specific documents are often indicators of potential non-conformance or risk. Addressing these issues today requires engineering teams to conduct additional tedious and error-prone manual investigation or data clarification. Incompleteness could indicate a regulatory requirement is not addressed or does not apply to a system, or it could be a simple omission. Inconsistencies and discrepancies can occur between regulatory documents as emerging technologies yield new standards and best practices that may be different from—or even conflict with—previous or overlapping regulations.

It is important to recognize that incompleteness, discrepancies, and inconsistencies may be ok in some contexts, *i.e.*, they are not always incorrect. In other cases, however, they indicate the need for additional investigation. In most cases today, however, identifying these issues is a lengthy human-intensive activity.

Software engineering teams in high-stakes domains are under mounting pressure to navigate the extensive maze of regulations and manage increasingly complex software systems without compromising on quality or safety. This challenge is compounded by the need to accelerate engineering and deployment processes to maintain a competitive edge against adversaries. It is therefore crucial for these teams to employ innovative tools that enhance efficiency through automation, particularly for frequent and time intensive tasks, such as identifying inconsistencies and discrepancies in documents.

Detecting inconsistencies and discrepancies within and between documents is a necessary and important first step. This process is not the same, however, as recommending solutions or resolving these issues automatically. Determining the applicability of LLMs to specific use cases requires assessing the opportunities and risks among multiple considerations for their contexts (Stephany Bellomo et al., 2023).

Given the mission-and safety-critical nature of government-acquired software systems, any automation should ensure that trained people remain central to the activities and confirm the safety and accuracy of the software engineering processes. This research area thus involves augmenting people in the software engineering activities primarily by *automating* DIID detection. Once DIID are detected, future work can develop solutions for corrective action. Entirely automating resolution of DIID via LLMs is not currently viable given the level of technology maturity and potential risks.

This chapter focuses on DIID detection in the software domain. However, the challenges and importance of DIID detection applies to many domains, including law, medical, government compliance, and education. Leveraging the ability of LLMs to examine extensive documents or data sets depends on accurate DIID detection to extend their benefits beyond merely assisting humans on highly specialized tasks or summarization. In software engineering and the SDLC, DIID detection is an important activity for scaling LLM use by improving consistency checking across artifacts, both within and across SDLC activities.

### *B. Definitions*

Document incompleteness, inconsistencies, and discrepancies all indicate potential problems with the content, but they differ in their specific implications, as follows:

- *Incompleteness* refers to important context or terms are missing in within or between documents. For example, if a software safety policy document states the software architecture must provide a detailed analysis of *safety-critical* signals, but a derived statement requires the software architecture to just provide a detailed analysis of *signals*, the "safety-critical" context is both important and missing. Incompleteness indicates that some requirements or analysis are missing or that the system context does not warrant specific requirements. In either case, additional human review and analysis are recommended to rectify incompleteness to reduce risk.
- An *inconsistency* refers to contradictions, lack of agreement/uniformity, or cohesion within a single document or across multiple documents. For example, safety and security are used synonymously (or with significant overlap) in some domains, while in other domains they imply two completely different set of requirements. Inconsistencies may not indicate a significant error but can create confusion to readers and reviewers. In addition, such inconsistencies may also suggest incompleteness.
- A *discrepancy* refers to a lack of compatibility or a divergence within or between documents related to facts that are either *direct* (*e.g.*, factual) or *derived* (*e.g.*, policy, narrative, or theoretical). For example, different response measures (0.01 sec versus 0.1 sec) listed for the same functionality in different components in a safety-critical system can result in fatal failures in software. Discrepancies can indicate a more significant error or issue, particularly if the information is used for confirmation or decisions.

These definitions are typical of what is found in dictionaries and literature on human interpretation or document analysis. Although there is no comprehensive list of different types of document incompleteness, inconsistencies, and discrepancies, common types and examples are summarized in Table 2 below.

***Table 2 Common Types of Incompleteness, Inconsistencies, and Discrepancies***

| Incompleteness | Inconsistency | Discrepancy |
| --- | --- | --- |
| • **Incomplete**: Important context or terms are missing. | • **Terminology**: Using different terms interchangeably without clear definitions or consistency.<br>• **Structural**: Lack of uniform structure in presenting information. | • **Factual discrepancies**: Conflicting factual information.<br>• **Policy or procedural discrepancies**: Deviations from established protocols.<br>• **Narrative discrepancies**: Different user stories fail to align.<br>• **Theoretical discrepancies**: Actual results conflict with theoretical predictions. |

### *C. A Software Engineering Safety Case Study*

The case study below provides additional context and illustrates DIID detection concepts in an R&D project we're conducting at the SEI. This case study involves a software team developing control software for a drone that must meet system and software safety requirements defined in government regulatory policies and guidance. In addition, the software team is applying industry standards, such as software safety analysis and secure coding, to enhance confidence in the software. These regulatory documents provide a starting point for examining DIID detection.

Figure 5 provides a software safety example, where it is important to detect DIID at every step between the policies, standards, and guidance documents (shown in the upper left) tracing through to the safety verification documents (shown in the lower right). DIID considerations related to the case study summarized below, with examples for each circled letter in Figure 5:

(A) **<u>Regulatory documents</u>** – DIID can occur between multiple policy, standards, and guidance documents. For example, a software safety standard states that safety requirements traceability must be performed and must follow a specific format, but a separate organizational safety policy states that the need for a software safety requirements traceability is flexible and says nothing about requiring a specific format.

(B) **<u>Safety Requirements</u>** – DIID can occur between regulatory documents and safety requirements document that lists safety requirements derived from the regulatory documents that are relevant to a specific system. For example, an engineering team identifies a set of 24 safety requirements, but one of the safety requirements states that a Failure Modes, Effects & Criticality Analysis (FMECA) (*Gargama, H. and Chaturvedi, S. K. 2011*) must be performed and always report any fault condition, but another regulatory document which also requires FMECA requires the system to always return to a safe state, not just report the fault.

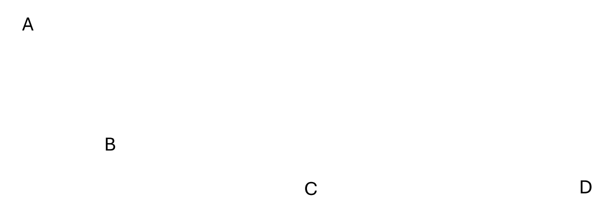


*Figure 5: A Software Engineering Safety Case Study.*

(C) **Derived Safety Requirements** – DIID can occur between safety requirements documents and derived safety requirements that extend the safety requirements from top level to every software module or component in the system. For example, in a software system with seven software modules, five of the modules have derived safety requirements that limit the use of dynamic memory allocation, but two modules have no such requirement.

(D) **Safety Verification** – DIID detection can occur with safety verification documents that summarize verification evidence (such as test results, analysis results, etc.) when compared to the derived safety requirements. For example, a derived safety requirement states a software module that processes three types of data messages must always reject improperly formatted messages (such as messages with potentially corrupted data). However, the safety verification information for that software module indicates that only one of the three message types was tested for rejecting improperly formatted messages.

The example shown in Figure 5 highlights the following four themes:

1. DIID detection requires **analysis of multiple large documents**. For example, for step A, the set of documents can be numerous (10+ documents) and many of the documents are 50 pages or more.
2. DIID detection can require **consideration of system specific context** or information. For example, for step B, the regulatory documents are general for any system, but the safety requirements are specific to one system.
3. DIID detection is an important consideration at every step, but sometimes at **different levels of abstraction**. For example, for step C, the safety requirements are a summary for the entire software system and may have 10-20 general software safety requirements, but the derived safety requirements are detailed for all software system components and will be a much larger list.
4. DIID detection can include comparison of **different types of information**. For example, for step D, the derived safety requirements are detailed requirements, but the safety verification information could be analysis results, test results, or other supporting evidence, including graphical information.

These themes provide insight into common limitations of current DIID approaches and suggest opportunities for using LLMs as part of new solutions to automate DIID detection.

### *D. Applying ChatGPT-4o to Detect Document Incompleteness, Inconsistencies, and Discrepencies*

We now build upon the software safety example shown in Figure 5 to explore DIID detection more thoroughly using a NASA software safety standard called the "SOFTWARE ASSURANCE AND SOFTWARE SAFETY STANDARD" (2022 NASA-STD-8739.8B). This NASA software safety standard is a publicly-available 65 page PDF file that defines "the requirements to implement a systematic approach to Software Assurance (SA), software safety, and Independent Verification and Validation (IV&V) for software created, acquired, provided, or maintained by or for NASA." It includes topics on software safety requirements and independent verification and validation that are similar to many government and commercial software safety documents.

For our case study, this NASA software standard is part of circle letter A in Figure 5. It describes software safety requirements that must be supported. In a large software-reliant acquisition program, this document would be one of many standards that the software must support, but we will just use one standard for this example.

With the software safety standard defined for circle A in Figure 5, consider a list of Safety Requirements in circle B. Requirements of this type are commonly expressed in word processing documents, spreadsheets, or a requirements management tool. There might be dozens of general safety requirement statements that derive from the standards in circle A. For example, a requirement from the NASA standard could include the following requirement from page 27 of the standard: "*The software detects inadvertent memory modification and recovers to a known safe state.*" This software safety requirement is derived directly from the standard and represents a top-level requirement of the software.

Continuing to trace through the steps in Figure 5, the Derived Safety Requirements in circle C are more specific decompositions of the requirements in circle B to specific parts of the system. Examining the memory requirement, there are multiple cases of potential "inadvertent memory modification" that should be considered. Some examples include:

- *Detecting accidental memory corruption from a software defect or bug*
- *Detecting memory modifications from a malicious virus that is running on the system and possibly changing memory*
- *Detecting memory modifications on the heap, the stack, or both.*

For our case study example, we consider the system as a drone or vehicle that must be resilient from accidental memory corruption but assert that malicious viruses will be prevented via cyber security practices and network protection mechanisms. With this system context, we identify the following Derived Safety Requirements for circle C:

- *The software must detect accidental heap memory corruption from a software defect or bug*
- *The software must return to a known safe state when accidental heap memory modification is detected*.

Tracing to the last step in Figure 5, the items in circle D are Safety Verification statements from the system/software developer that provide evidence that software safety requirements are met. Documents in this circle D group include architecture documents, design documents, code, and software test results. For example, a design document provided by software developers might have the following statement about the software:

- *The software design includes a "System Monitor' component that detects heap memory modifications and, when detected, returns the system to a known safe state.*

Using the information from each stage of Figure 5, we applied ChatGPT-4o to detect DIID via a browser interface. In particular, we uploaded the NASA 8739.8 pdf file and included the text prompt shown in Figure 6 to ChatGPT-4o.

> I will provide a list of statements, from software requirements and development documents, and I want to detect where the statements are inconsistent, incomplete, or there are discrepancies as compared to the NASA 8739.8 software standard. Please be specific in your responses and provide supporting rationale for the answers.
>
> The Safety Requirements for the software include the following statement:
> - The software detects inadvertent memory modification and recovers to a known safe state.
>
> The Derived Requirements for the software include the following statements:
> - The software must detect accidental heap memory corruption from a software defect or bug
> - The software must return to a known safe state when accidental heap memory modification is detected.
>
> The Safety Verification documents from the software developer including the following statement:
> - The software design includes a "System Monitor' component that detects heap memory modifications and, when detected, returns the system to a known safe state.
>
> Analyze these statements as compared to the NASA 8739.8 standard and identify where the statements are inconsistent, incomplete, or there are discrepancies as compared to the NASA 8739.8 software standard.

***Figure 6: ChatGPT-4o Prompt to Detect DIID.***

Within seconds, ChatGPT-4o provided the informative response shown in Figure 7.

To assess the provided statements against the NASA-STD-8739.8 Software Assurance and Software Safety Standard, let's examine the consistency, completeness, and alignment of each statement.

**Safety Requirements**

- "The software detects inadvertent memory modification and recovers to a known safe state."

This aligns with the NASA-STD-8739.8 safety-critical software requirements that mandate software must:

- Detect, report, and take corrective action if the system reaches a potentially hazardous state.
- Initialize to a known safe state on startup or after restarts.

The statement meets the standard broadly but lacks explicit details about the mechanisms for detection and recovery.

**Derived Requirements**

1. "The software must detect accidental heap memory corruption from a software defect or bug."
2. "The software must return to a known safe state when accidental heap memory modification is detected."

Both derived requirements address specific scenarios of memory corruption and recovery. However:

- These requirements narrow the scope to heap memory corruption, potentially excluding other types of inadvertent memory modifications (e.g., stack or global memory) that could lead to unsafe states. The standard requires a broader coverage of detecting any hazardous memory condition.

**Safety Verification Document**

- "The software design includes a 'System Monitor' component that detects heap memory modifications and, when detected, returns the system to a known safe state."

This statement aligns well with the derived requirements but does not fully encompass the broader safety requirement. The focus is on heap memory modifications, which may not cover all potential inadvertent memory issues. Furthermore:

- There is no explicit mention of how the "System Monitor" achieves these actions, which is necessary for verification against the standard's emphasis on traceability and hazard mitigation.

**Identified Inconsistencies and Gaps**

1. Incompleteness of Scope:
   The safety requirement covers all inadvertent memory modifications, but the derived requirements and verification statements only address heap memory. This discrepancy could lead to insufficient coverage of other critical memory areas.

2. Verification Specificity:
   The verification statement describes a system monitor but lacks details about testing and validation, which are required to ensure compliance with the standard's rigorous safety verification processes.

3. Traceability and Testing:
   The derived requirements and safety verification document should include bi-directional traceability to show alignment with safety-critical requirements. This traceability ensures completeness in identifying and mitigating all hazards associated with memory modifications.

*Figure 7: ChatGPT4o DIID Detection Response.*

ChatGPT-40's response in Figure 7 shows how it first analyzed and compared the NASA standard to each of the statements from each stage shown in Figure 5, including Safety Requirements, Derived Requirement's, and Safety Verification Documents. In each stage, ChatGPT-4o provided a summary about DIID along with some comments. For example, in the Derived Safety Requirements, ChatGPT-4o identified the statements "address specific scenarios of memory corruption and recovery" and then states that the requirements "narrow the scope" compared to the NASA standard. These statements are correct and the ChatGPT-4o response is the type of insight that can aid system and software engineers reviewing these statements.

In addition to the response above, ChatGPT-4o also provided the recommendations shown in Figure 8. Not only are these recommendations accurate, they also provide keen insights into the level of evidence and data that provides confidence of software safety, at least with respect to memory corruption. These types of recommendations can help software developers and certification authorities provide a data driven and independent view of software safety information.

> **Recommendations**
>
> 1. Expand the derived requirements to include all types of inadvertent memory modifications (heap, stack, and global memory) to align with the broader scope of the safety requirement.
> 2. Provide detailed descriptions in the safety verification document about how the "System Monitor" functions and include validation methodologies.
> 3. Ensure traceability between the safety requirement, derived requirements, and verification steps to satisfy the traceability criteria outlined in NASA-STD-8739.8.
> 4. Include specific test cases and results for memory corruption scenarios, demonstrating the system's capability to detect and recover from various types of memory issues.
>
> By addressing these gaps, the requirements and verification processes will better align with NASA's software safety standards. Let me know if you'd like to refine any of these points or need additional guidance.

***Figure 8: ChatGPT-4o DIID Recommendations.***

Current methods of detecting DIID are time-intensive and require humans to both find and analyze these statements at each stage to ensure software safety. This safety requirement example for Figure 5 shows how ChatGPT-4o can assist humans to detect DIID quickly in standards and policy documents for additional analysis. LLMs are becoming more capable and accurate, and this example demonstrates the vital role LLMs will have assisting software engineers. The analysts, who is assumed to be an expert in safety critical requirements and regulations, can use this information to locate related areas in documents and rectify them.

This example demonstrates how LLMs can detect DIID and provide useful assistance to human software engineers. However, it also hints at areas of caution for software engineering and the essential role of experienced software engineers. For example, while ChatGPT-40's recommendations are accurate compared to the NASA standard, human engineers should decide if some requirements do not apply to the system context.

This case study example also applies LLMs to areas that play to the strengths of LLMs. For example, determining what the decomposed software safety requirements should be for a specific system context is an important analysis that should be driven by humans with a review process, and using LLMs could be high risk if the LLM is incorrect in generating the requirements. However, using LLMs to detect DIID has lower risk because the LLMs are analyzing information we provide that yields better and more accurate LLM responses.

### *E. Related Work on DIID Detection*

Related work to advance DIID detection can be grouped into document analysis, prompt engineering and prompt patterns, and human analysis, as discussed below.

*1) Document Analysis.* Document analysis research is a relevant related work area for DIID detection in software engineering. This analysis is a common topic of interest across many domains, including healthcare (Shokrollahi et al., 2023) (Moilanen et al., 2022), financial or business environment (Cao et al., 2024; Han et al., 2023; Shukla et al., 2023) and for analyzing legal documents (Bauer et al., 2023; Deroy et al., 2023; Prasad et al., 2024). Document analysis is used in most of these domains both to provide summaries and insights into data for a given domain. Some approaches include building domain-specific models (Wu et al., 2023), though other approaches use existing models and attempt to address the scale of documents or establish measures for effectiveness.

A primary area of research for DIID detection is a response to the issue of generated document summaries being inconsistent with the original document (Laban et al., 2023; Lattimer et al., 2023; Luo et al., 2023; Tang et al., 2023). These discussions confirm the importance of consistency in document summarization and analysis. These reports also highlight the risks of using these systems in production use cases, including a medical domain example that underscores serious risks from inconsistencies (Tang et al., 2023).

A recent paper proposes a benchmark called SUMMAC that includes proposed measurement and inconsistency detection datasets to address inconsistencies in summaries (Laban et al., 2021). Inconsistency checks between an original document and a summarized document is a simpler use case and differs from DIID detection across multiple and different documents. However, the discussions on evaluation and testing data sets are promising and were assessed as part of our DIID detection work.

Another related area of research is detecting hallucinations in LLM-generated output, not limited only to document summaries (Fallah et al., 2024; Rawte et al., 2024). To address this challenge, some solutions are white-box approaches that require detection mechanisms within the model (Zhu et al., 2024), though this approach is not suitable for DIID detection in software engineering for average users. For example, LLM fine tuning approaches can refine LLM responses with additional data, but may alter the LLM responses in unintended ways. Black-box solutions for DIID detection (*i.e.*, where there is no requirement to look inside an LLM) are much easier for users to apply, but are limited to constraints of the LLM developers such as being limited to smaller documents.

A limitation in current document analysis approaches is evaluation of "long" documents (Cao et al., 2024; Pradeep et al., 2023). Although the definition of "long document" is subjective, there are multiple summarization successes for short text, such as news summaries (Barta et al., 2024; Goyal et al., 2023). Proposed approaches for analyzing larger documents include chunking (Schwaber-Cohen, n.d.), architectural approaches using Retrieval-Augmented Generation (RAG) to enable LLM integration with large data sets (Gao et al., 2024; Ke et al., 2024; *Retrieval Augmented Generation (RAG) for LLMs – Nextra*, n.d.), multi-agent-based approaches (Zhao et al., 2024), fine tuning or modification of the models to adjust sequence length (Chen et al., 2023; Ding et al., 2023), building new models (Cohan et al., 2018), and hybrid approaches that give LLMs access to metadata about document (Saad-Falcon et al., 2023). New testing frameworks and datasets are also necessary to assess the effectiveness in long documents (C. Wang et al., 2024).

Of these multiple approaches to support long documents or larger data sets, RAG-based and agent-based approaches are particularly applicable to DIID due to their architectural benefits. For example, key architectural qualities needed for DIID in software engineering projects are (1) *scalability*, which supports larger and larger data sets, and (2) *extensibility*, which supports more data types over time, beyond documents. RAG-based and agent-based approaches provide this flexibility, although RAG retrieval accuracy is dependent upon multiple factors, including the data storage approach (Zhang, T.).

*2) Prompt Engineering and Prompt Patterns.* The flexibility of LLMs to support a wide range of uses effectively is supported in part by prompt engineering, which involves structured interactions with—and programming of—LLM computational systems to solve complex problems via natural language interfaces (P. Liu et al., 2023). Prompt patterns (White, Fu, et al., 2023) codify best practices for phrasing prompts to maximize extraction accuracy and provide knowledge transfer mechanisms to problem-solve with LLMs more accurately. Prompt patterns also enable more effective and repeatable performance of LLMs, and many patterns have been identified for a range of task objectives (White, Hays, et al., 2023).

Prompt engineering is a key capability that provides new opportunities for human interactivity with documents and data for DIID detection in software engineering. For example, the *Context Manager* pattern (White, Fu, et al., 2023) directs an LLM to set the context, which is one of the four themes identified the *A Software Engineering Safety* in Section IV.C above. Prompt engineering, including identification and refinement of relevant prompt patterns, is thus central to our DIID research. When DIID types are narrowly defined and examples are provided to LLMs through fine tuning or prompt engineering, DIID detection can be improved for specific use cases (Hegselmann et al., 2024).

*3) Human Analysis.* The primary means for performing DIID detection is currently people, so considering research beyond the field of computer science helps characterize DIID's current effectiveness and limitations. One recent example (Schoor et al., 2023) conducted an experiment with ~160 college students who read two short texts, some with consistent information and others with inconsistent information, and then wrote a short essay. This experiment explored reading behaviors and outcomes, and it highlighted the following points relevant for software acquisition:

- When people encounter discrepancies in information their "attention to sources" increased. This finding is relevant to software acquisition because their discussion also indicated that context and document sources are important in DIID detection.
- Users of any solution will require more information about the sources behind the finding before accepting—or perhaps even considering—any suggestion from an AI-augmented tool about DIID detection.

*4) Testing and Evaluation Frameworks.* Testing frameworks that can assess LLM stengths and weaknesses for detecting DIID are needed to provide consistent means of assessing the many available LLMs and versions. A proposed testing protocol for DIID for document summaries (Laban et al., 2023) is promising because it provides tests specific to DIID and benchmarks for recent LLMs. This work provides a useful DIID detection test framework and dataset because it considers multiple discrepancy examples, with multiple LLMs, and compares results of a few benchmarks. It also highlights the importance—and many challenges—of creating an effective testing framework, particularly because accurate labeling of test data is hard to scale. However, their approach—like previous approaches—is limited to only a subset of the DIID detection needed for software acquisition and doesn't cover all the levels of abstraction shown in Figure 4.

A related new area of research is Generative Informational Retrieval (Gen-IR), which focuses on building more effective testing of generative AI-augmented systems for use cases like document summarization and grounded answer generation

(Arabzadeh & Clarke, 2024; Bénédict et al., 2023). This research area is promising since it provides testing data sets and baselines that may be relevant to DIID detection. However, the testing bias of prompts and the limited scaling of human validation remains a challenge for this research area.

## V. Concluding Remarks

This chapter highlighted opportunities to employ generative AI—specifically large language models (LLMs)—to a wide range of software development lifecycle (SDLC) activities. We explored the transformative potential of LLMs in software engineering, focusing on how these models can enhance various SDLC phases. We covered both the benefits and challenges of integrating LLMs (such as their ability to accelerate tasks like code generation, testing, and documentation), while also noting common LLM risks (such as generating incorrect or incomplete outputs).

As one example scenario, we explored using LLMs to improve software verification and compliance with regulatory standards. Using a software safety case study, we explored how the opportunities and risks of applying LLMs in the SDLC can be addressed and integrated into new SDLC workflows. We demonstrated using LLMs to detect document incompleteness, inconsistencies, and discrepancies (DIID) in software safety requirements and artifacts in the context of a NASA software safety standard.

The following are key observations and ongoing areas of study in conducting the DIID analysis work described in this chapter:

- ***LLM-driven DIID detection can help ensure compliance in high-stakes domains*** – LLMs are maturing to the point where they can be applied to detect and rectify document incompleteness, inconsistencies, and discrepancies (DIID) in software-reliant systems. DIID detection is increasingly important in high-stakes (and thus often regulated) domains, such as healthcare, aerospace, and defense. This chapter provided a foundation for understanding the capabilities and future directions of DIID detection, which are critical to ensure compliance and minimize risks associated with non-conformance to regulations.
- ***Leveraging vector and agent-based LLM architectures enhances document analysis efficiency*** – The emerging role of LLMs augmented with vector database and/or agent-based architectures have demonstrated promising results that enhance the efficiency and accuracy of DIID detection. By leveraging semantic similarity across texts, LLMs offer an innovative approach to handling the expansive and complex documentation inherent in government projects. Moreover, the introduction of prompt engineering as a means to customize LLM interactions via prompt patterns (White, Fu, et al., 2023) enables new tailored automation, aligning closely with the specific requirements and frameworks of software acquisition.
- ***Collaboration between developers and LLMs is essential for effective software engineering*** – It's unrealistic to expect today's LLMs to generate complete and flawless software-reliant systems from scratch. Instead, they should be viewed through the lens of generative *augmented* intelligence (AI+), where developers work together with AI-augmented tools. We are applying this type of collaboration in our research, teaching, and programming (*e.g.,* by working hand-in-hand with LLMs like ChatGPT, Gemini, and Claude), but don't expect them to generate all software artifacts. Instead, we do much of the design, decomposition, and some of the implementation tasks, and then have LLMs augment us and our teammates with tasks otherwise tedious and/or error-prone to perform manually. We thus use LLMs to *supplement* our skills, rather than to *supplant* them, recognizing that the effective use of LLMs requires a combination of prompt engineering and human oversight to maximize benefits and minimize risks.

Our work to date provides a basis for additional research in the following areas that extend the potential benefits for software acquisition and potentially into other domains:

- ***Recommending resolutions*** – Our future work will focus on the semantic and contextual information used in DIID detection to suggest relevant recommendations for resolving identified issues. We also expect prompt engineering will provide humans with effective methods and tools to gain insight into specific contexts and codify prompt patterns to resolve DIID detections. Future research should also include testing frameworks and benchmarks that characterize and enable validation of provided DIID recommendations.
- ***Extending into AI-Augmented software engnieering*** – The example shown in ***Figure 5*** can be extended beyond safety verification to include inconsistency checks against various software engineering artifacts, for example to assist requirements elicitation. The application of LLMs to software engineering and related artifacts is an active area of research (Carleton et al., 2021, 2023; Ozkaya et al., 2023).
- ***Ethical considerations*** – Bias and fairness are known challenges with LLMs, so future research should consider the impact of these issues in the context of SDLC processes. Research concerning fairness in document summarization (Zhang et al., 2023) provides insights on how to characterize and possibly mitigate these issues.

Looking ahead, the opportunity for further research on AI-augmented DIID detection is both extensive and enticing. The potential to expand the scope of LLM applications to encompass recommendation systems for resolving DIID detections or to integrate LLMs more deeply into other stages of software engineering underscores the potential impact of this research. Further research is needed to refine the accuracy of detection mechanisms, enhance the utility of LLMs for non-specialist users, and extend the capabilities to more complex and diverse document sets. By streamlining software acquisition processes, these advances can safeguarding the integrity and efficacy of mission- and safety-critical applications and operations in high-stakes domains. Integrating LLMs across the SDLC also has the potential to aid humans by enhancing multiple software engineering activities, increasing the reliability and efficiency of software testing and deployment processes, and enabling humans to focus on the most challenges aspects of software development, thereby resulting in more stable and robust software releases.

## ACKNOWLEDGMENTS

TBD.

## REFERENCES

Abbas, A. (2023, June 12). 5 Ways LLMs Can Empower Software Engineering. *Techopedia*. https://www.techopedia.com/5-ways-llms-can-empower-software-engineering

Abdali, S., Anarfi, R., Barberan, C., & He, J. (2024, March 19). *Securing Large Language Models: Threats, Vulnerabilities and Responsible Practices*. https://www.semanticscholar.org/paper/Securing-Large-Language-Models%3A-Threats%2C-and-Abdali-Anarfi/9a741f33aa4d782639e1f81a7e9c341b58b6ed2a

*Amplify · GitHub*. (n.d.). https://github.com/orgs/gaiin-platform/repositories

Arabzadeh, N., & Clarke, C. L. A. (2024). *A Comparison of Methods for Evaluating Generative IR* (No. arXiv:2404.04044). arXiv. http://arxiv.org/abs/2404.04044

*AutoGen | AutoGen*. (n.d.). https://microsoft.github.io/autogen/

Barstow, D. (1987). Artificial intelligence and software engineering. *Proceedings of the 9th International Conference on Software Engineering*, 200–211.

Barta, B., Lakatos, D., Nagy, A., Nyist, M. K., & Ács, J. (2024). *From News to Summaries: Building a Hungarian Corpus for Extractive and Abstractive Summarization* (No. arXiv:2404.03555). arXiv. https://doi.org/10.48550/arXiv.2404.03555

Bauer, E., Stammbach, D., Gu, N., & Ash, E. (2023). *Legal Extractive Summarization of U.S. Court Opinions* (No. arXiv:2305.08428). arXiv. http://arxiv.org/abs/2305.08428

Bénédict, G., Zhang, R., & Metzler, D. (2023). *Gen-IR @ SIGIR 2023: The First Workshop on Generative Information Retrieval* (No. arXiv:2306.02887). arXiv. http://arxiv.org/abs/2306.02887

Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., & Gazit, I. (2023). Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue*, *20*(6), Pages 10:35-Pages 10:57. https://doi.org/10.1145/3582083

Bright, J., Enock, F. E., Esnaashari, S., Francis, J., Hashem, Y., & Morgan, D. (2024). *Generative AI is already widespread in the public sector* (No. arXiv:2401.01291). arXiv. https://doi.org/10.48550/arXiv.2401.01291

Cao, T., Raman, N., Dervovic, D., & Tan, C. (2024). *Characterizing Multimodal Long-form Summarization: A Case Study on Financial Reports* (No. arXiv:2404.06162). arXiv. https://doi.org/10.48550/arXiv.2404.06162

Carleton, A., Klein, M., Robert, J., & Harper, E. (2021). *Architecting the Future of Software Engineering: A National Agenda for Software Engineering Research & Development*. Software Engineering Institute. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=741193&_gl=1*1oddlfa*_ga*mtq5mzi1mtcumty2njczmjc2ng..*_ga_87wecw6hcs*mty2njczmjc2ns4xljeumty2njczmjc3my4wljauma..

Carleton, A., Shull, F., Schmidt, D. C., Harper, E., Robert, J., & Ozkaya, I. (2023, August 25). *U.S. Leadership in Software Engineering & AI Engineering: Critical Needs & Priorities Workshop - Executive Summary*. https://insights.sei.cmu.edu/library/us-leadership-in-software-engineering-ai-engineering-critical-needs-priorities-workshop-executive-summary/

Chen, S., Wong, S., Chen, L., & Tian, Y. (2023). *Extending Context Window of Large Language Models via Positional Interpolation* (No. arXiv:2306.15595). arXiv. https://doi.org/10.48550/arXiv.2306.15595

Churilla, M., Vanhoudnos, N., & Beveridge, R. (2023, May 15). *The Challenge of Adversarial Machine Learning*. https://doi.org/10.58012/jrjp-n210

Cohan, A., Dernoncourt, F., Kim, D. S., Bui, T., Kim, S., Chang, W., & Goharian, N. (2018). A Discourse-Aware Attention Model for Abstractive Summarization of Long Documents. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, 615–621. https://doi.org/10.18653/v1/N18-2097

*Crosstalk, The Journal of Defense Software Engineering—APAN Community*. (n.d.). https://community.apan.org/wg/crosstalk/

Deroy, A., Ghosh, K., & Ghosh, S. (2023). *How Ready are Pre-trained Abstractive Models and LLMs for Legal Case Judgement Summarization?* (No. arXiv:2306.01248). arXiv. https://doi.org/10.48550/arXiv.2306.01248

Dhar, R., Vaidhyanathan, K., & Varma, V. (2024). *Can LLMs Generate Architectural Design Decisions? -An Exploratory Empirical study* (No. arXiv:2403.01709). arXiv. https://doi.org/10.48550/arXiv.2403.01709

Ding, J., Ma, S., Dong, L., Zhang, X., Huang, S., Wang, W., Zheng, N., & Wei, F. (2023). *LongNet: Scaling Transformers to 1,000,000,000 Tokens* (No. arXiv:2307.02486). arXiv. http://arxiv.org/abs/2307.02486

Fallah, P., Gooran, S., Jafarinasab, M., Sadeghi, P., Farnia, R., Tarabkhah, A., Taghavi, Z. S., & Sameti, H. (2024). *SLPL SHROOM at SemEval2024 Task 06: A comprehensive study on models ability to detect hallucination* (No. arXiv:2404.04845). arXiv. https://doi.org/10.48550/arXiv.2404.04845

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2024). *Retrieval-Augmented Generation for Large Language Models: A Survey* (No. arXiv:2312.10997). arXiv. https://doi.org/10.48550/arXiv.2312.10997

Gargama, H. and Chaturvedi, S. K. (2011) "Criticality Assessment Models for Failure Mode Effects and Criticality Analysis Using Fuzzy Logic," in IEEE Transactions on Reliability, vol. 60, no. 1, pp. 102-110, doi: 10.1109/TR.2010.2103672.

Gärtner, A. E., & Göhlich, D. (2024). Automated requirement contradiction detection through formal logic and LLMs. *Automated Software Engineering*, *31*(2), 49. https://doi.org/10.1007/s10515-024-00452-x

*Generative AI Capability Model*. (n.d.). Retrieved September 13, 2024, from https://www.ibm.com/architectures/hybrid/genai-capability-model

Robert J. & Schmidt D. (2023) *Generative AI Q&A: Applications in Software Engineering*. (2023, November 16). https://insights.sei.cmu.edu/blog/generative-ai-question-and-answer-applications-in-software-engineering/

Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X., & Liao, X. (2024). *Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning*. 453–465. https://www.computer.org/csdl/proceedings-article/icse/2024/021700a453/1WDJaRUZRKg

Goyal, T., Li, J. J., & Durrett, G. (2023). *News Summarization and Evaluation in the Era of GPT-3* (No. arXiv:2209.12356). arXiv. https://doi.org/10.48550/arXiv.2209.12356

Grand, G., Wong, L., Bowers, M., Olausson, T. X., Liu, M., Tenenbaum, J. B., & Andreas, J. (2024). *LILO: Learning Interpretable Libraries by Compressing and Documenting Code* (No. arXiv:2310.19791; Version 4). arXiv. https://doi.org/10.48550/arXiv.2310.19791

Gurunathan, S. (2024, January 30). *Using LLMs to Automate Pipeline Conversions From Legacy to Tekton*. DevOps.Com. https://devops.com/using-llms-to-automate-pipeline-conversions-from-legacy-to-tekton/

Han, Y., Hou, J., & Sun, Y. (2023). Research and Application of GPT-Based Large Language Models in Business and Economics: A Systematic Literature Review in Progress. *2023 IEEE International Conference on Computing (ICOCO)*, 118–123. https://doi.org/10.1109/ICOCO59262.2023.10397642

Harrison, T. H., Levine, D. L., & Schmidt, D. C. (1997). The design and performance of a real-time CORBA event service. *SIGPLAN Not.*, *32*(10), 184–200. https://doi.org/10.1145/263700.263734

Hegselmann, S., Shen, S. Z., Gierse, F., Agrawal, M., Sontag, D., & Jiang, X. (2024). *A Data-Centric Approach To Generate Faithful and High Quality Patient Summaries with Large Language Models* (No. arXiv:2402.15422). arXiv. http://arxiv.org/abs/2402.15422

*How to put generative AI to work in your DevSecOps environment*. (n.d.). Retrieved September 13, 2024, from https://about.gitlab.com/blog/2024/03/07/how-to-put-generative-ai-to-work-in-your-devsecops-environment/

Jasmin Jahic, Ashkan Sami (2024) State of Practice: LLMs in Software Engineering and Software Architecture. ICSA-C 2024: 311-318

Jiang, E., Olson, K., Toh, E., Molina, A., Donsbach, A., Terry, M., & Cai, C. J. (2022). PromptMaker: Prompt-based Prototyping with Large Language Models. *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, 1–8. https://doi.org/10.1145/3491101.3503564

Jin, M., Shahriar, S., Tufano, M., Shi, X., Lu, S., Sundaresan, N., & Svyatkovskiy, A. (2023). InferFix: End-to-End Program Repair with LLMs. *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1646–1656. https://doi.org/10.1145/3611643.3613892

John Robert & Douglas Schmidt. (2024, January 22). *10 Benefits and 10 Challenges of Applying Large Language Models to DoD Software Acquisition*. https://insights.sei.cmu.edu/blog/10-benefits-and-10-challenges-of-applying-large-language-models-to-dod-software-acquisition/

Ke, Y., Jin, L., Elangovan, K., Abdullah, H. R., Liu, N., Sia, A. T. H., Soh, C. R., Tung, J. Y. M., Ong, J. C. L., & Ting, D. S. W. (2024). *Development and Testing of Retrieval Augmented Generation in Large Language Models—A Case Study Report* (No. arXiv:2402.01733). arXiv. http://arxiv.org/abs/2402.01733

Heiko Koziolek, Anne Koziolek (2024) LLM-based Control Code Generation using Image Recognition. LLM4CODE@ICSE 2024: 38-45

Laban, P., Kryściński, W., Agarwal, D., Fabbri, A. R., Xiong, C., Joty, S., & Wu, C.-S. (2023). *LLMs as Factual Reasoners: Insights from Existing Benchmarks and Beyond*. https://doi.org/10.48550/ARXIV.2305.14540

Laban, P., Schnabel, T., Bennett, P. N., & Hearst, M. A. (2021). *SummaC: Re-Visiting NLI-based Models for Inconsistency Detection in Summarization* (No. arXiv:2111.09525). arXiv. https://doi.org/10.48550/arXiv.2111.09525

Lattimer, B., CHen, P., Zhang, X., & Yang, Y. (2023). Fast and Accurate Factual Inconsistency Detection Over Long Documents. In H. Bouamor, J. Pino, & K. Bali (Eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (pp. 1691–1703). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.emnlp-main.105

Li, L., Yang, L., Jiang, H., Yan, J., Luo, T., Hua, Z., Liang, G., & Zuo, C. (2022). AUGER: Automatically generating review comments with pre-training models. *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1009–1021. https://doi.org/10.1145/3540250.3549099

Liu, P., Zhang, L., & Gulla, J. A. (2023). *Pre-train, Prompt and Recommendation: A Comprehensive Survey of Language Modelling Paradigm Adaptations in Recommender Systems* (No. arXiv:2302.03735). arXiv. https://doi.org/10.48550/arXiv.2302.03735

Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., & Wang, Q. (2024). Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 1–13. https://doi.org/10.1145/3597503.3639180

Liu, Z., Yang, Z., Liao, Q. 2024 "Exploration On Prompting LLM With Code-Specific Information For Vulnerability Detection," 2024 IEEE International Conference on Software Services Engineering (SSE), pp. 273-281, doi: 10.1109/SSE62657.2024.00049.

Lones, M. (2024, March 20). LLMs and the future of programming languages [Substack newsletter]. *Fetch Decode Execute*. https://fetchdecodeexecute.substack.com/p/llms-and-the-future-of-programming

Luo, Z., Xie, Q., & Ananiadou, S. (2023). *ChatGPT as a Factual Inconsistency Evaluator for Text Summarization* (No. arXiv:2303.15621). arXiv. https://doi.org/10.48550/arXiv.2303.15621

Moilanen, T., Sivonen, M., Hipp, K., Kallio, H., Papinaho, O., Stolt, M., Turjamaa, R., Häggman-Laitila, A., & Kangasniemi, M. (2022). Developing a Feasible and Credible Method for Analyzing Healthcare Documents as Written Data. *Global Qualitative Nursing Research*, *9*, 23333936221108706. https://doi.org/10.1177/23333936221108706

2022 NASA-STD-8739.8B. Software Assurance and Software Safety Standard [Review of Software Assurance and Software Safety Standard]. NASA Standard. https://standards.nasa.gov/standard/NASA/NASA-STD-87398

Nguyen, S. (n.d.). Council Post: Revolutionizing Software Development With Large Language Models. Forbes. Retrieved December 9, 2024, from https://www.forbes.com/councils/forbestechcouncil/2024/03/20/revolutionizing-software-development-with-large-language-models/

Ozkaya, I. (2023a) The next frontier in software development: AI-augmented software development processes *IEEE Software* 40 (4), 4-9

Ozkaya, I. (2023b) Can architecture knowledge guide software development with generative AI? *IEEE Software* 40 (5), 4-8

Ozkaya, I., Carleton, A., Robert, J., & Schmidt, D. C. (2023, October 2). *Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?* https://doi.org/10.58012/6n1p-pw64

Pan, R., Ibrahimzada, A. R., Krishna, R., Sankar, D., Wassi, L. P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R. (2024). Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. ICSE 2024: 82:1-82:13

Panyam, S. (2024). *Council Post: Software Engineering Challenges In The Age Of AI: The Role Of Strong Engineering Leadership*. Forbes. Retrieved September 2, 2024, from https://www.forbes.com/councils/forbestechcouncil/2024/06/24/software-engineering-challenges-in-the-age-of-ai-the-role-of-strong-engineering-leadership/

Pradeep, R., Hui, K., Gupta, J., Lelkes, A. D., Zhuang, H., Lin, J., Metzler, D., & Tran, V. Q. (2023). *How Does Generative Retrieval Scale to Millions of Passages?* https://doi.org/10.48550/ARXIV.2305.11841

Prasad, N., Boughanem, M., & Dkaki, T. (2024). *Exploring Large Language Models and Hierarchical Frameworks for Classification of Large Unstructured Legal Documents* (No. arXiv:2403.06872). arXiv. https://doi.org/10.48550/arXiv.2403.06872

Purba, M.D., Ghosh, A., Radford, B.J., & Chu, B. (2023). Software Vulnerability Detection using Large Language Models. 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), 112-119.

Rawte, V., Tonmoy, S. M. T. I., Rajbangshi, K., Nag, S., Chadha, A., Sheth, A. P., & Das, A. (2024). *FACTOID: FACtual enTailment fOr hallucInation Detection* (No. arXiv:2403.19113). arXiv. https://doi.org/10.48550/arXiv.2403.19113

*Retrieval Augmented Generation (RAG) for LLMs – Nextra*. (n.d.). https://www.promptingguide.ai/research/rag#rag-evaluation

Robert, J., Ivers, J., Schmidt, D. C., Ozkaya, I., & Zhang, S. (2024). The Future of Software Engineering and Acquisition with Generative AI. *Crosstalk, AI Taming the Beast*, 26–43. https://community.apan.org/wg/crosstalk/m/documents/464157

Saad-Falcon, J., Barrow, J., Siu, A., Nenkova, A., Yoon, D. S., Rossi, R. A., & Dernoncourt, F. (2023). *PDFTriage: Question Answering over Long, Structured Documents* (No. arXiv:2309.08872). arXiv. https://doi.org/10.48550/arXiv.2309.08872

Sami, M. A., Rasheed, Z., Waseem, M., Zhang, Z., Herda, T., & Abrahamsson, P. (2024). *Prioritizing Software Requirements Using Large Language Models* (No. arXiv:2405.01564). arXiv. https://doi.org/10.48550/arXiv.2405.01564

Schoor, C., Rouet, J.-F., & Britt, M. A. (2023). Effects of context and discrepancy when reading multiple documents. *Reading and Writing*, *36*(5), 1111–1143. https://doi.org/10.1007/s11145-022-10321-2

Schwaber-Cohen, R. (n.d.). *Chunking Strategies for LLM Applications | Pinecone*. https://www.pinecone.io/learn/chunking-strategies/

*SEI CERT Coding Standards—CERT Secure Coding—Confluence*. (n.d.). Retrieved October 23, 2024, from https://wiki.sei.cmu.edu/confluence/display/seccode

Sherman, M. (2024). Advances in Using LLMs for Securing Software [Review of Advances in Using LLMs for Securing Software]. https://app.swapcard.com/widget/event/ai4-2024/planning/UGxhbm5pbmdfMTkwMzA2NA==

Shokrollahi, Y., Yarmohammadtoosky, S., Nikahd, M. M., Dong, P., Li, X., & Gu, L. (2023). *A Comprehensive Review of Generative AI in Healthcare*. https://doi.org/10.48550/ARXIV.2310.00795

Shukla, N. K., Katikeri, R., Raja, M., Sivam, G., Yadav, S., Vaid, A., & Prabhakararao, S. (2023). Investigating Large Language Models for Financial Causality Detection in Multilingual Setup. *2023 IEEE International Conference on Big Data (BigData)*, 2866–2871. https://doi.org/10.1109/BigData59044.2023.10386558

Srinivasan, K., & Fisher, D. (1995). Machine learning approaches to estimating software development effort. *IEEE Transactions on Software Engineering*, *21*(2), 126–137. IEEE Transactions on Software Engineering. https://doi.org/10.1109/32.345828

Stephany Bellomo, Shen Zhang, James Ivers, Julie B. Cohen, & Ipek Ozkaya. (2023). *Assessing Opportunities for LLMs in Software Engineering and Acquisition*. Carnegie Mellon University Software Engineering Institute. https://doi.org/10.58012/m3hj-6w28

Tang, L., Sun, Z., Idnay, B., Nestor, J. G., Soroush, A., Elias, P. A., Xu, Z., Ding, Y., Durrett, G., Rousseau, J., Weng, C., & Peng, Y. (2023). *Evaluating Large Language Models on Medical Evidence Summarization*. https://doi.org/10.1101/2023.04.22.23288967

*The MITRE AI Maturity Model and Organizational Assessment Tool Guide*. (2023). https://www.mitre.org/news-insights/publication/mitre-ai-maturity-model-and-organizational-assessment-tool-guide

Tikayat Ray, A., Cole, B., Pinon Fischer, O., Bhat, A. P., White, R., & Mavris, D. (2023). Agile Methodology for the Standardization of Engineering Requirements Using Large Language Models. *Systems*, *11*, 352. https://doi.org/10.3390/systems11070352

*Using ChatGPT to Analyze Your Code? Not So Fast*. (2024, February 12). https://insights.sei.cmu.edu/blog/using-chatgpt-to-analyze-your-code-not-so-fast/

*Using LLMs in Software Requirements Specifications: An Empirical Evaluation*. (n.d.). Retrieved September 13, 2024, from https://arxiv.org/html/2404.17842v1#bib.bib17

Wang, C., Duan, H., Zhang, S., Lin, D., & Chen, K. (2024). *Ada-LEval: Evaluating long-context LLMs with length-adaptable benchmarks* (No. arXiv:2404.06480). arXiv. https://doi.org/10.48550/arXiv.2404.06480

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2023). *Software Testing with Large Language Model: Survey, Landscape, and Vision* (No. arXiv:2307.07221). arXiv. http://arxiv.org/abs/2307.07221

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT* (No. arXiv:2302.11382). arXiv. https://doi.org/10.48550/arXiv.2302.11382

White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2023). *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design* (No. arXiv:2303.07839). arXiv. https://doi.org/10.48550/arXiv.2303.07839

Wu, S., Irsoy, O., Lu, S., Dabravolski, V., Dredze, M., Gehrmann, S., Kambadur, P., Rosenberg, D., & Mann, G. (2023). *BloombergGPT: A Large Language Model for Finance* (No. arXiv:2303.17564). arXiv. https://doi.org/10.48550/arXiv.2303.17564

Xia, C. S., Wei, Y., Zhang, L. (2023) Automated Program Repair in the Era of Large Pre-trained Language Models. ICSE 2023: 1482-1494

Zhang, T., Yan, B., & Jaffri, A. (2024). How to Improve and Optimize Retrieval- Augmented Generation Systems [Review of How to Improve and Optimize Retrieval- Augmented Generation Systems]. In www.gartner.com. Gartner.

Zhang, Y., Zhang, N., Liu, Y., Fabbri, A., Liu, J., Kamoi, R., Lu, X., Xiong, C., Zhao, J., Radev, D., McKeown, K., & Zhang, R. (2023). *Fair Abstractive Summarization of Diverse Perspectives* (No. arXiv:2311.07884). arXiv. https://doi.org/10.48550/arXiv.2311.07884

Zhao, J., Zu, C., Xu, H., Lu, Y., He, W., Ding, Y., Gui, T., Zhang, Q., & Huang, X. (2024). *LongAgent: Scaling Language Models to 128k Context through Multi-Agent Collaboration* (No. arXiv:2402.11550). arXiv. https://doi.org/10.48550/arXiv.2402.11550

Zhu, D., Chen, D., Li, Q., Chen, Z., Ma, L., Grossklags, J., & Fritz, M. (2024). *PoLLMgraph: Unraveling Hallucinations in Large Language Models via State Transition Dynamics* (No. arXiv:2404.04722). arXiv. https://doi.org/10.48550/arXiv.2404.04722