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 & Mary
Williamsburg, VA, USA

ABSTRACT

Large Language Models (LLMs) are a form of generative AI that have become engines of 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.

This chapter addresses the question of whether the transformative changes associated with LLMs will yield more efficient software development methods and tools and create different software engineering workflows with meaningful improvements or not? To answer this question, we review the use cases of using LLMs on common software engineering activities and combine these insights to discuss the potential impact of LLMs across the SDLC.

Keywords: Large language models, LLM, generative AI, AI-augmented software engineering and software acquisition, software development lifecycle (SDLC).

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, 2023).

This chapter examines when—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.

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;; and Section V presents concluding remarks and outlines future research and advancements.

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.[1] 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).

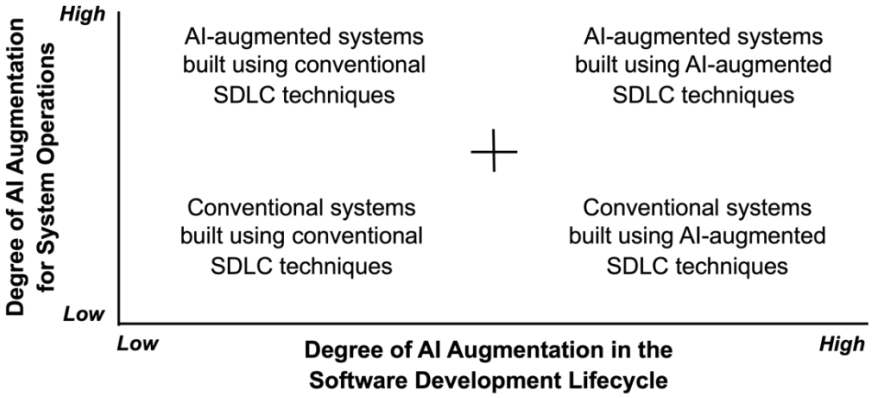

Figure 1: Taxonomy of AI Augmentation for System Operations & Software Development Lifecycle Activities.

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).

[1] DevSecOps stands for "development, security, and operations" and it provides an integrated approach and toolset to deliver incremental software updates while considering the security implications throughout the update.

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 for 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 activities, 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.

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.

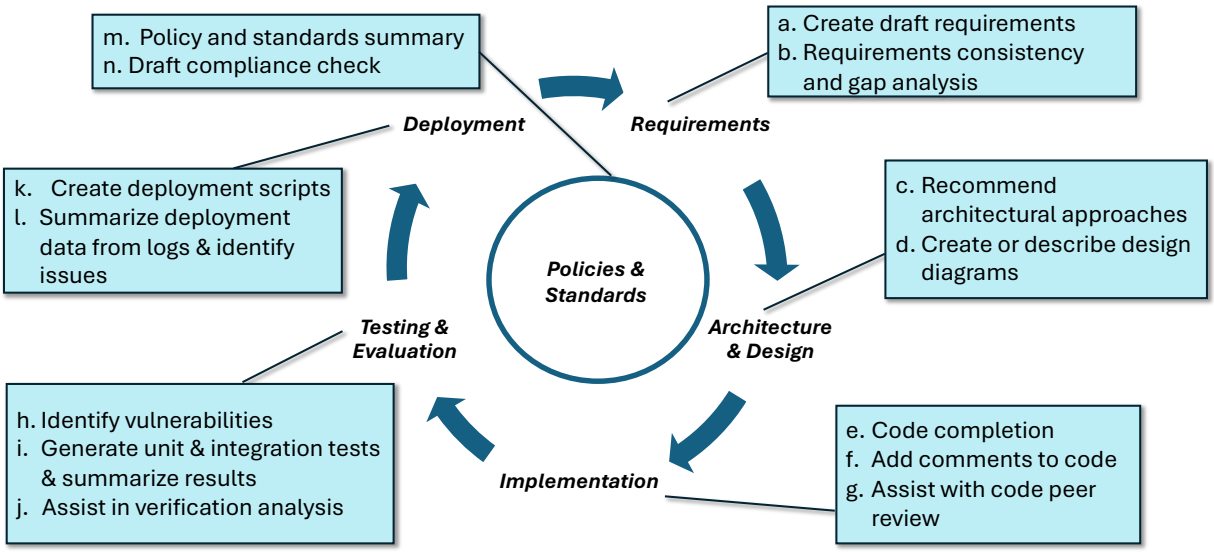

Figure 2: Opportunities to Apply LLMs to Software Development Lifecycle (SDLC) Activities.

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 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 2023). 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 in generating diagrams of 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 DevSecOps 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 many 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 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 (Freeman et al., 2025a, Freeman et al., 2025b). 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

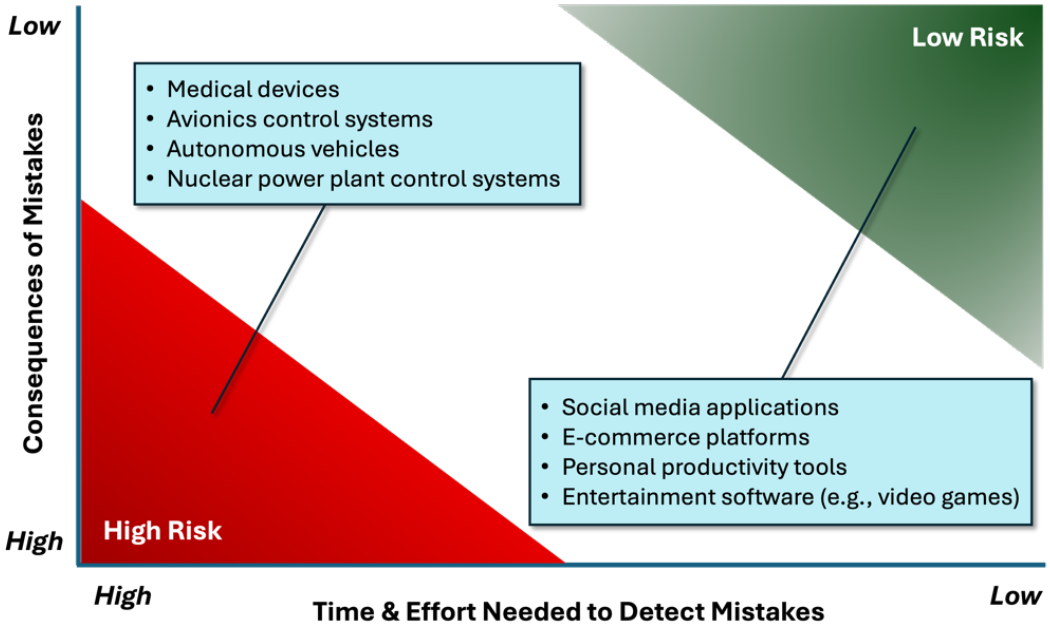

Figure 3: Evaluating Risks Associated with Applying LLMs to Some Representative Application Domains.

(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 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**.[2]

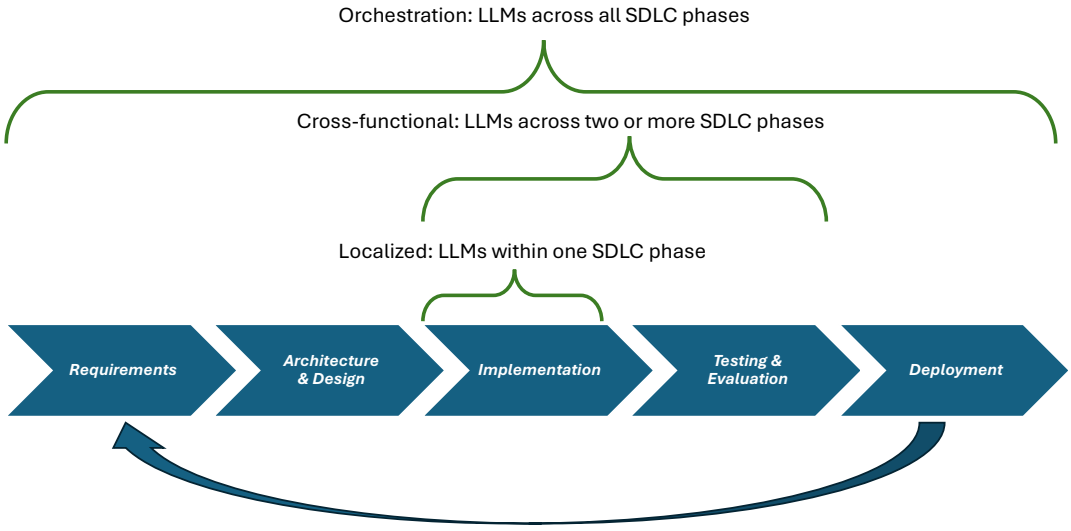

Figure 4: Layers of LLM Application to SDLC

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 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. 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).

Inserting LLMs into the SDLC for software teams can be viewed from multiple perspectives. There are many established frameworks for examining and enabling technology insertion or iterative improvements in software development activities, each offering different perspectives on adoption and integration. Some examples include the following:

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

- The Technology Acceptance Model (TAM) focuses on usefulness and ease of use, helping developers assess whether a new tool will enhance workflow.
- Diffusion of Innovation Theory categorizes adopters into groups (innovators, early adopters, majority, laggards) and explains how new technologies gradually gain acceptance.
- Agile Change Management aligns technology adoption with Agile workflows, ensuring seamless integration through iterative feedback and adaptation.
- The DevOps CALMS Framework (Culture, Automation, Lean, Measurement, Sharing) emphasizes continuous improvement, fostering collaboration and technical advancements in development practices.

Given the rapid pace of advancement of LLMs and their capabilities across the SDLC, an approach that focus on incremental insertion in specific SDLC phases to accomplish a desired objective is a good approach. Starting with small teams, or individuals, can also be helpful and provide a foundation to explore specific risks and opportunities. For example, consider an organization that does software development in multiple systems or products, and a small team for a legacy product continues to be costly to sustain due to limited market and budget. This software context is a common example in industry and government organizations and forces teams to find higher efficiency in software updates to address bugs or new software security fixes.

In this example, LLMs might enable efficiency in the Implementation and Testing and Evaluation phases to enable timely (and correct) code updates followed by additional testing to ensure the updates are correct. This application of LLMs could take the form of engineers using a chat interface or using a software development tool specific to the activity, such as testing. In this example, a new workflow would be created, improved, and then potentially shared with other teams in the organization. As more LLM tools become available, it is expected that some LLM tools will target this type of software context and provide an overarching LLM Cross-Functional or even orchestration solution.

As we emphasized in this paper, many SDLC activities will require hybrid tools and humans with targeted support from generative AI to accomplish a shift in their effective execution. Accomplishing this transition successfully first requires understanding the nature of these activities and where LLMs can be applied. Software architecture is an example of such exploration, while some architectural activities enable effective incorporation of generative AI into SDLC workflows, others don't (Ivers and Ozkaya, 2025). Any first step in investigating where to start applying LLMs in the SDLC should start with understanding the decomposition of activities into tasks, aligning subtasks with the strengths of generative AI, and designing the new workflows with a collaborative approach that includes generative AI, other AI, additional automated tools, and human experts.

Many such explorations are already progressing rapidly in software engineering and anecdotal and empirical research is emerging that evaluates the capabilities available LLMs and codifies various success and failure examples. This research—including examples we describe in this paper—is crucial for effective use of LLMs as an effective partner in software engineering. The key gap, however, is our understanding of what being a useful partner may mean. Software engineers and LLMs accomplishing software-reliant system development can take many forms depending on the nature of the SDLC activities. Software engineering research must also make progress in exploring the forms of this partnership and provide exemplar effective approaches, as well as anti-patterns, to establish robust LLM-supported workflows.

ACKNOWLEDGMENTS

Thanks to the reviewers of this book chapter for providing suggestions that greatly improved its form and content.

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

Bellomo, S., Zhang, S., Ivers, J., Cohen, J., & Ozkaya, I. (2023). *Assessing Opportunities for LLMs in Software Engineering and Acquisition*. Carnegie Mellon University Software Engineering Institute. https://doi.org/10.58012/m3hj-6w28

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://insights.sei.cmu.edu/documents/1308/2021_014_001_741195.pdf.

Freeman, L., Schmidt, D.C., Bonnell, A., Robert, J., and Wojton, H. (2025a) "AIRC Panel on Generative AI in the Acquisition Lifecycle," (YouTube webinar https://www.youtube.com/watch?v=ZlCc94w-2bY), Acquisition Innovation Research Council.

Freeman, L., Robert, J., and Wojton, H. (2025b) "The Impact of Generative AI on Test & Evaluation: Challenges and Opportunities," in Proceedings of the International Workshop on Envisioning the AI-Augmented Software Development Lifecycle, Trondheim, Norway, June 2025.

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

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/

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

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

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

Ivers, J, Ozkaya, I (2025) Will Generative AI Fill the Automation Gap in Software Architecting? New and Emerging Ideas, 22nd ICSA 2025, Odense, Denmark.

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, 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

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

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

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.

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

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

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==

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

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/

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

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