The Rise of Context Engineering: How Generative AI Can Transform Systems Engineering

Scott Gallant
Independent Researcher
Orlando, Florida, United States of America
Scott@EffectiveApplications.com

Chris McGroarty
United States Army Combat Capabilities Development Command
Orlando, Florida, United States of America
christopher.j.mcgroarty.civ@army.mil

Abstract— **The emergence of Large Language Models (LLMs) into software development challenges current systems engineering practices. Methods built around prescriptive documentation, hierarchical decomposition, and artifact-driven traceability were designed for human interpretation but provide limited value to LLMs, which depend on structured, narrative-rich context rather than diagrams or complex requirement documents. Previous adaptations of Agile and model-based approaches improved human collaboration but did not address how to convey intent and domain understanding to generative models. Improving the utility of LLMs can completely transform software engineering with very exciting benefits including increasing productivity, error reduction, knowledge amplification, and innovation. Feeding LLMs with human-centric systems engineering artifacts or simple prompts has resulted in poorly implemented software. This paper introduces *Context Engineering* as a conceptual framework for systematically curating and organizing contextual information to guide AI-augmented software development. The central idea is that context-rich artifacts such as user stories, scenarios, and knowledge embeddings can serve as the foundation for more accurate, aligned, and trustworthy LLM-generated software systems.**

Keywords—context engineering, systems engineering, software development, LLMs, testing

I. Introduction

The discipline of systems engineering has historically been characterized by structured processes that emphasize completeness, traceability, and alignment between requirements, design, and verification. To ensure this alignment and to manage the complexity of large systems, these processes rely heavily on formal artifacts such as requirements documents and design models. Many industries use formal architectural frameworks, such as the Department of Defense Architecture Framework (DoDAF) [1] or the NATO Architecture Framework (NAF) [2]. Most systems engineering processes and frameworks include graphical modeling languages such as Unified Modeling Language (UML) [3] and System Modeling Language (SysML) [4]. These frameworks, formats, and methods are useful with human-managed software programs because they help people coordinate and communicate with a shared language.

The introduction of Large Language Models (LLMs) [5] into the software development process amplifies the need for concise communication of user requirements. LLMs are not traditional developers because they do not consume requirements in the same manner as human engineers, nor do they have all the context or the users. Instead, they work on prompts given by the user that are usually limited and inefficient to generate software. Their effectiveness depends on the richness and clarity of the prompt and the context with which the model was trained.

Using LLMs to generate software introduces a paradigm shift where the discipline of systems engineering must move from producing prescribed ambiguous artifacts to curating contextual scaffolding. This emerging discipline is called Context Engineering [6] and is concerned with the systematic elicitation, structuring, and management of contextual information that enables both humans and Artificial Intelligence (AI) systems to align on intent, constraints, and operational meaning. Context Engineering can leverage concepts from modern systems engineering philosophies like Agile [7] including user stories and scenario descriptions to communicate intent in ways that generative AI can better interpret.

The purpose of this paper is threefold. First, it covers the limitations of legacy systems engineering approaches when applied to LLM-augmented software development. Second, it articulates the basis for Context Engineering as a successor paradigm, emphasizing its reliance on narrative-rich artifacts and continuous refinement of context. Third, it explores methodological implications for engineering teams adapting their processes to incorporate LLMs as better collaborators.

II. Systems Engineering Evolution

Systems engineering can be considered an art and a science of describing a system's requirements and design while removing ambiguity for the purposes of improving engineering outcomes. Systems engineering for software has a decades-long history of practice, large formalized organizations for standards and best practices, and thousands of books and papers on producing the most ideal artifacts to build systems. There are many aspects and some complex approaches and sub-disciplines like systems thinking, interdisciplinary approaches, life cycle management, and risk management. Systems engineering can be summarized as creating and managing information artifacts that allow engineers to build exactly the intended systems.

Software engineering has undergone a shift from structured and tedious documents that are difficult to manage toward agile methodologies with iterative collaboration and development for more responsiveness to change. User stories, backlogs, and sprint reviews represent a departure from rigid documentation toward lightweight narrative forms of specification, which are designed to provide enough context for teams to engineer what the users need. Agile methods embody a recognition that in dynamic environments, adaptability and shared understanding between users, systems engineers, and developers is more valuable than formal structure and completeness.

This evolution unintentionally aligns with the needs of LLMs and Context Engineering, where structured but flexible artifacts such as user stories and scenarios can be more easily and accurately consumed and used than large requirements documents and architecture design models.

III. LLM Operation and Limitations

As we evolve to using LLMs for more software development, we need to understand how LLMs work, how they are optimized, and what their limitations are in order to use them most effectively in our goals of software development.

Large language models (LLMs) process text by first tokenizing input into discrete units and converting these tokens into numerical representations called embeddings. These representations are then passed through layers of a transformer architecture, where attention mechanisms capture contextual relationships. The model can then generate output based on probabilistic predictions of tokens based on learned patterns [5].

Current LLMs have inherent issues that limit their effectiveness in software engineering. They operate on the immediate context of the input prompt and do not maintain long-term state, which can lead to inconsistencies in large systems. While LLMs excel at recognizing patterns in code, they do not truly understand program semantics, which can result in syntactically correct but logically flawed or inefficient software. Ambiguities in context (requirements, design, and domain-specific knowledge) further reduce their reliability.

IV. Towards Context Engineering

Context Engineering is the emerging discipline concerned with the systematic gathering, structuring, and management of contextual information. This can provide an evolution in software development in environments where LLMs serve as implementors. Context Engineering is more than prompt engineering, which relies on a finite context window of strings as model inputs. Context engineering is a more systematic and modular approach where context is dynamically assembled from components and interfaces. This approach enhances scalability and enables evolving context.

A. *Evolvability of Software and Systems Engineering Artifacts*

The objective of using Context Engineering for software engineering is to reframe information and processes to ensure that intent, constraints, and dependencies are captured in forms that both humans and AI can operationalize across iterative cycles of design, development, and validation. It is important that the engineering artifacts remain human readable and mutable so that the human remains in control of an evolving software codebase. Evolvability is a critical property of modern software systems, ensuring that they can be adapted, extended, and maintained as requirements and environments change. Normalized Systems Theory (NST) [8] provides a theoretical foundation for this principle, demonstrating that software architectures must be designed to avoid combinatorial effects that inhibit long-term change. In the context of Context Engineering, this requirement becomes even more important because as LLMs are introduced into the development process, the curated context that guides their outputs must support continuous adaptation without degradation of coherence or traceability. Context Engineering is not only about supplying sufficient information for generative tasks, but also about structuring that information in ways that preserve the evolvability of the resulting systems, consistent with the principles of NST.

B. *Artifacts of Context Engineering*

This section includes some practical examples of Context Engineering artifacts that achieve the goals of human readability/editability and context-rich information for LLMs to generate quality software. These include structured user stories with acceptance criteria to capture intent, operational scenarios that ground functionality in real-world contexts, and architecture narratives that provide technical design intent. Complementing these are coding standards and design principle guides that ensure stylistic and architectural consistency, as well as evolution logs that document key decisions and their rationale over time. Finally, reusable prompt templates and interaction patterns establish consistent ways of engaging with LLMs. Together, these artifacts form a contextual scaffold that is accessible to humans, consumable by LLMs, and adaptable as systems evolve.

User stories are popular and useful artifacts that describe intended system capabilities from the user's perspective and usually have the format of "As a <type of user>, I want <capability> so that <reason/value> [7]. As input to an LLM, a user story can simply be structured in YAML [9] to remain human readable while easily consumed by LLMs [10]:

```
User Story: As a registered user, I want to reset my password
via email so that I can regain access to my account.
Acceptance Criteria:
  - User can request password reset with a valid email.
  - System generates a secure token valid for 15 minutes.
  - Email is sent with reset link and instructions.
```

Contextual scenarios can provide more information beyond requirements. These can inform LLMs about workflows, real-world usage, and edge cases:

```
For a heavy-volume scenario, 100,000 users attempt to login
simultaneously. The system should provide reactive responses
to inputs, throttle requests gracefully and in a scalable manner,
and retain session integrity of all user sessions.
```

System architecture context can provide some important framing and structure for the LLM to work within. This can be useful when operating in environments with strict security, architecture, or any other implementation requirements. This also gives the human designer both the ability to direct the LLM inside important guardrails as well as the ability to quickly change those guardrails as technologies, organizations, or requirements evolve.

```
System Context: The application is composed of three
services: API Gateway, Authentication Service, and Database.
- API Gateway handles routing and load balancing.
- Authentication Service issues JWT tokens.
- Database stores user credentials with bcrypt encryption.
```

Design guides and coding styles are often created by organizations for more consistent implementations of software so the development team can operate with the same understanding leading to easier code reviews, modifications, and implied guidance from senior developers for less errors. Although difficult to deploy and maintain with human software developers, these guidelines can be more easily enforced with LLMs. Here is a short example which should be elaborated on before using:

```
Coding Standards:
- Use Java 21 Long-Term Support.
- Use the five design principles of SOLID: Single
Responsibility Principle, Open/Closed Principle, Liskov
Substitution Principle, Interface Segregation Principle, and
Dependency Inversion Principle.
- Use the UpperCamelCase naming convention for class and
interface names.
- Use camelCase for method and variable names.
- Use JavaDocs for public classes, methods, and other
significant code blocks to explain their function and usage.
- Choose names that clearly communicate the purpose of the
variable, class, interface, or method.
```

Evolution logs can serve as a context journal with a running record of decisions, assumptions, and rationale for changes. This can help both humans and LLMs avoid repeating the same mistakes.

```
Decision Log – 2025-09-14
- Switched from SQLite to PostgreSQL due to scaling
concerns.
- Context: Anticipated growth to 100k users by 2026.
- Implications: Update deployment scripts and DB schemas.
```

Prompt templates can be specifically designed for a chosen LLM and help establish consistency for recurring actions. Researching, optimizing and evolving prompts for a specific LLM (per context windows, reasoning ability, role complexity, and specific formatting) can improve quality of outputs. These prompts being used by a large team of engineers can both save time and improve consistency across a large team.

C. *Visual Artifacts – Reversing the Workflow*

Visual systems engineering diagrams such as sequence diagrams and architecture diagrams help the human brain better understand ideas and contextualize complicated elements of software design. In practice, these diagrams created by human systems engineers can often be ambiguous or lacking enough detail for software developers. Although there have been advancements in LLM ability to interpret visual diagrams, the reliance on visual diagrams to be contextually complete means relying on both humans and visual formats that are flawed or limited in the level of detail necessary to generate software. Even well-formed UML or SysML diagrams are limited in detail, can be unwieldy to maintain, and can be proprietary to the modeling tools used.

To mitigate these challenges, the workflow should be reversed so that the LLM is generating the visual diagrams based on textual inputs to communicate with the human engineers based on the same context that the LLM used to generate software. These textual inputs provide more coherent and complete context that the LLM can reliably interpret [11]. The generated diagrams then serve as human-readable artifacts for design communication, verification, and collaborative discussion, rather than as primary inputs for the LLM's reasoning or code generation. This approach preserves the benefits of visual modeling while avoiding reliance on limited diagram comprehension capabilities of LLMs.

V. Introducing LLMs to Existing Software Projects

With engineers knowledgeable in LLMs and Context Engineering, an organization can set up a process, systems engineering content, and a set of prompt templates on new software projects. The setup of a systems engineering process and agreed upon artifacts is required whether an organization is using LLMs for software development or human software engineers so any additional cost of Context Engineering would be low. For well-established software projects with a large amount of systems engineering artifacts and existing software, the cost to transition to Context Engineering and using an LLM could appear intimidating, but there are methods that could help make this transition.

Systems engineering artifacts including requirements documents, design models, interface specifications, test cases, as well as software in existing code repositories can be very large and seem difficult to transition to Context Engineering. In reality, these products can be a rich source of domain knowledge that can be leveraged to fine-tune LLMs and larger projects could benefit the most [12].

Artifacts can first be converted into machine-readable, structured formats and segmented into semantically coherent units such as individual requirements, classes, methods, or design components. Each unit is then encoded into a vector representation using an embedding model, which captures both the content and its contextual relationships within the artifact. Metadata, including artifact type and version, can be incorporated to further enrich the embeddings. These structured embeddings, which are semantically rich [13], are stored in a vector database, allowing a Retrieval-Augmented Generation (RAG) framework [14] to dynamically provide relevant context to the LLM during code generation or documentation tasks. By leveraging retrieved structured embeddings, the LLM can reason over historical project knowledge without requiring full memorization, ensuring alignment with existing software and systems engineering artifacts. As new artifacts are produced or updated, embeddings can be refreshed, creating a continuously synchronized, context-aware LLM application that maintains consistency and scales with large projects.

VI. Process Implications

All software projects require configuration management for the software as well as the systems engineering artifacts that guide development. These artifacts (e.g. requirements documents, UML diagrams, and design models) usually exist in heterogeneous formats, many of which are binary, tool-specific, and difficult to version effectively, especially the relationships between the artifacts. The introduction of Context Engineering artifacts for LLM-enabled development fundamentally shifts these configuration management requirements. Instead of managing a diverse set of formats and complicated relationships between the content in those files, configuration management in the Context Engineering approach can focus on textual representations of user stories, scenarios, design guidelines, and related contextual data.

A. *Data-Driven Systems Engineering by Context Engineering*

Context Engineering can have a transformative impact on our past work towards a data-driven, generative simulation ecosystem. The simplification of systems engineering artifacts from heterogenous, and in some cases binary formats into text, enables the use of lightweight version control systems and databases to track changes, compare revisions, and support collaborative workflows.

The structured textual format also allows a data-driven systems engineering approach that the military simulation domain has attempted without LLMs for several years. We developed a database-driven systems engineering tool that generated software, tests, and systems engineering artifacts using domain specific information in a relational database and templating technology [15,16] before LLMs were created. Domain Specific Languages (DSLs) [17] can be used to normalize nouns and verbs within a project, across multiple projects in a domain, and across standards organizations. We have been working towards a DSL for the military simulation domain [18] across the North Atlantic Treaty Organization (NATO) Science and Technology Organization (STO). Now with LLMs and Context Engineering, a machine understanding of domain concepts (our nouns and verbs within Context Engineering textual artifacts) can provide context across projects, standards organizations, and industries improving generative software for everybody involved.

B. *Deterministic Generation and Reliability*

LLMs are probabilistic systems by design and generate outputs based on learned token distributions rather than rule-based semantics, which limits their ability to produce software in a strictly deterministic manner comparable to compilers. However, near-deterministic behavior can be achieved by carefully constraining their operation through controlled inputs and context. This includes fixing decoding parameters (e.g., setting temperature to zero for greedy decoding), providing structured and comprehensive prompts, retrieving coherent engineering artifacts through RAG, and embedding generation within strict templates aligned with system requirements. While the LLM itself remains probabilistic, embedding it within a structured engineering pipeline transforms the overall process into a more predictable system for practical purposes, thereby enabling reliable software generation in artifact-rich development environments. To ensure reliability, software generation should also be coupled with automated verification processes, such as static code analysis and automated testing.

C. *Testing*

Verification and Validation (V&V) [19], or testing of systems within the context of user requirements requires a full understanding of the requirements and context when analyzing the performance of a system. This is a time-consuming and error-prone activity that machines can now automate and perform more accurately if the machine has the full context of the system, including those artifacts used throughout the Context Engineering and generative software development process. Here are three types of LLM related tools for testing.

There has been a lot of progress in using LLMs for testing. NVIDIA created the Hephaestus (HEPH) framework [20] that automates test-case generation using LLMs. It indexes requirements, interface descriptions, architecture documents, and code samples in an embedding database, retrieves relevant fragments, and generates test specifications and implementations (executable test software). The tests are compiled, executed, and coverage data is collected, with feedback loops generating new cases to close gaps. HEPH supports multi-format inputs and integrates with knowledge management tools like Confluence [21] and JIRA [22].

Playwright MCP [23] is a Model Context Protocol (MCP) server that exposes browser automation capabilities to AI agents. Agents can navigate pages, interact with Document Object Model (DOM) elements in the web page, run assertions, and generate tests from natural language scenarios. Playwright MCP uses structured browser data rather than pixel-based input, making tests more deterministic and reliable.

Finally, Replit Agent 3 [24] is a popular generative AI system that creates software systems based on prompts. It can generate software from scratch or examine existing software created elsewhere and make requested changes. Replit Agent 3 includes the ability to test its own work, make improvements, and retest. This testing includes creating user accounts, logging in, and using natural language text from the software creation prompt to test the functionality of software.

These types of tools will likely improve, but can perform better now by using domain-specific LLM improvements, well done Context Engineering artifacts, and a set of data-driven relationships across the systems engineering information.

VII. Conclusion and Future Work

This paper presented the systems engineering perspective of using LLMs for software generation and the need to transform from using human-focused requirements and design artifacts to more context-rich information optimized for LLMs to use for software generation. The concept of context engineering was detailed specifically for software generation. This included some proposed products including user stories, contextually rich scenarios, architectural context, and design guidance. Some guidance for introducing context engineering to existing large projects was provided. Then, the process implications for our software development were discussed including our data-driven systems engineering requirements and testing frameworks that could support our V&V requirements. Our future work will be to steadily introduce LLMs into our software development projects while transitioning our systems engineering methods from traditional V-model methods towards a data-rich context engineering methodology.

As LLMs continue to evolve in capability, scale, and integration into software engineering workflows, it is insufficient for systems engineering to remain static. The methods by which LLMs are employed must co-evolve with the models themselves. Early applications of LLMs in software development have largely mirrored existing practices, treating them as accelerators of human tasks within established paradigms. As LLMs assume more sophisticated roles transitioning from assistants to collaborators, and potentially to autonomous developers, the systems engineering processes need to evolve to optimize accuracy and timelines of development.

The emerging discipline of Context Engineering cannot be conceived merely as a downstream reaction to the technical trajectory of LLMs. Rather, it should develop in parallel and in dialogue with the creators of LLMs. Best practices in Context Engineering (how context is represented, structured, and managed for LLMs) should not only adapt to the capabilities of current LLM architectures but also inform the ongoing design of LLMs. The systems engineering community has both an opportunity and a responsibility: to contribute proactively to the shaping of LLM development, ensuring that the tools and the practices co-develop toward greater alignment, reliability, and contextual fidelity.

Acknowledgment

We would like to thank the United States Army Combat Capabilities Development Command Soldier Center for providing the resources and support that made this research possible. Their commitment to advancing innovation in modeling and simulation greatly contributed to this work.

References

[1] United States Department of Defense (DoD) Chief Information Officer (CIO), "The DoDAF Architecture Framework Version 2.02," August 2010. [Online]. Available: https://dodcio.defense.gov/Library/DoD-Architecture-Framework/

[2] North Atlantic Treaty Organization (NATO), "NATO Architecture Framework v4," January 2018. [Online]. Available: https://www.nato.int/nato_static_fl2014/assets/pdf/2021/1/pdf/NAFv4_2020.09.pdf

[3] Object Management Group (OMG), "OMG Unified Modeling Language (OMG UML) Version 2.5.1," December 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[4] Object Management Group (OMG), "OMG System Modeling Language Version 2.0 beta 4," July 2025. [Online]. Available: https://www.omg.org/spec/SysML

[5] A. Vaswani, et al., "Attention is all you need," Advances in Neural Information Processing Systems, 2017.

[6] L. Mei, et al., "A survey of context engineering for large language models," arXiv preprint arXiv:2507.13334, 2025.

[7] B. Douglass, Agile Systems Engineering. Morgan Kaufmann, 2015.

[8] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[9] YAML Language Development Team, "YAML Ain't Markup Language (YAML) version 1.2, revision 1.2.2," October 2021. [Online]. Available: https://yaml.org/spec/1.2.2/

[10] J. He, et al., "Does prompt formatting have any impact on LLM performance?," arXiv Novemer, 2024. arXiv:2411.10541.

[11] T. Topcu, M. Husain, M. Ofsa, and P. Wach, "Trust at your own peril: a mixed methods exploration of the ability of large language models to generate expert-like systems engineering artifacts and a characterization of failure modes," The Journal of The International Council on Systems Engineering, Volume 28, Issue 5, pp583-604, 2025.

[12] J. Shin, et al., "Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code," 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), pp. 490-502 in Ottawa, ON, Canada, 2025.

[13] A. Kozlowski, "Semantic structure in large language model embeddings," arXiv preprint, August, 2025. arXiv:2508.10003v1.

[14] P. Lewis, et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," Advances in Neural Information Processing Systems 33. 2020.

[15] S. Gallant and C. Gaughan, "Systems engineering for distributed live, virtual, and constructive (LVC) simulation," Proceedings of the 2010 Winter Simulation Conference, Baltimore, MD, 2010.

[16] S. Gallant, C. Metevier, and C. Gaughan, "Systems engineering and executable architecture for M&S," M&S Journal, Spring 2014.

[17] M. Mernik, J. Heering, and A. Sloane, "When and how to develop domain-specific languages." ACM computing surveys 37, no. 4, 2005.

[18] C. McGroarty, et al., "Creating robust evolvable MSaaS services: an integrated model-driven engineering approach," I/ITSEC, 2023.

[19] J. Elele, et al., "M&S requirements and VV&A requirements: what's the relationship?." ITEA Journal of Test & Evaluation vol. 37, issue 4, p333, 2016.

[20] Y. Dong, et al., "A survey on code generation with LLM-based agents," arXiv July, 2025. arXiv:2508.00083v1.

[21] E. Kalelioğlu, Implementing Atlassian Confluence: Strategies, Tips, and Insights to Enhance Distributed Team Collaboration Using Confluence. Packt Publishing Ltd, 2023.

[22] J. Fisher, D. Koning, and A. Ludwigsen, "Utilizing Atlassian JIRA for large-scale software development management," LLNL-CONF-644176. Lawrence Livermore National Lab.(LLNL), Livermore, 2013.

[23] Microsoft, "Introduction to Playwright MCP," 2025. [Online]. Available: https://playwright.dev/agents

[24] Replit, "Agent 3. Autonomy for All," 2025. [Online]. Available: https://replit.com/agent3