ETIS WHITE PAPER SERIES

WP-009

Context Engineering

Designing the Information, Memory, and Control Environment for Intelligent Systems

Core Thesis
Context is not background material supplied to an intelligent system. It is a designed, governed, versioned, and observable control environment that determines what the system can know, infer, decide, and do.

William T. O'Connell, Ph.D.
Adjunct Professor of Computer Science, Loyola University Chicago
Former IBM Distinguished Engineer and CTO, IBM Consulting Quality and Delivery
Creator, Engineering Trustworthy Intelligent Systems (ETIS)

July 2026
ETIS WP-009 | Version 1.0

Executive Abstract

The rapid improvement of large language models has made it tempting to treat intelligence as a property of the model. In deployed systems, however, model capability is only one factor. The quality and safety of an intelligent system depend heavily on the information it receives, the instructions that govern it, the tools it may invoke, the memory it retains, the identities and permissions surrounding it, and the evidence available to verify its behavior. These elements form the system's context environment.

Context engineering is the discipline of designing that environment. It extends beyond prompt engineering and retrieval-augmented generation. It includes requirements, architecture, repository knowledge, policies, examples, state, memory, data provenance, tool contracts, identity, authority, telemetry, and human escalation. It also includes the mechanisms that select, compress, prioritize, update, and remove context over time. Anthropic defines context engineering as curating and maintaining the optimal set of information during inference; AWS describes it as dynamically assembling the information an LLM needs for a task; GitHub increasingly treats repository instructions and agent-specific guidance as persistent engineering context. [1][2][3]

The central argument of this paper is that context engineering is becoming a core software-engineering discipline. As systems move from conversational assistance toward agentic execution, context becomes part of the control plane. Incorrect, stale, overbroad, manipulated, or untraceable context can produce wrong decisions and unauthorized actions even when the underlying model performs as designed. Conversely, well-engineered context improves consistency, reviewability, security, cost, and operational trust.

This paper develops a professional model for context engineering across the full lifecycle. It examines context architecture, repository-centered knowledge, retrieval, memory, protocols such as the Model Context Protocol and Agent2Agent, security and trust boundaries, evaluation, observability, governance, and organizational maturity. It concludes with the ETIS position: context must be engineered with the same rigor applied to code, data, interfaces, and operational controls.

> **Executive Takeaway**
> Prompt quality influences one interaction. Context architecture influences the behavior of the entire intelligent system.

1. From Prompt Engineering to Context Engineering

Prompt engineering emerged as a practical response to the sensitivity of language models to instructions and examples. It remains useful, but it frames the problem too narrowly. A production system does not receive only a prompt. It receives system instructions, conversation history, retrieved documents, user identity, application state, tool definitions, tool results, policies, examples, memory, and environmental signals. The system may also receive context assembled by another model or agent. The resulting behavior depends on how these elements interact, not merely on the wording of a single instruction.

The industry is therefore moving toward context engineering. Anthropic describes the discipline as selecting and maintaining the optimal set of tokens during inference, while emphasizing that more context is not always better. Long context can create distraction, contradiction, and reduced attention. [1] AWS similarly defines context engineering as designing systems that dynamically assemble the information needed for a task. [2] OpenAI's work on long-running engineering agents reports that repository knowledge must be structured as a map rather than a massive instruction manual. [4]

This shift changes the engineering question. The question is no longer only, “What should the model be told?” It becomes, “What information should be available, from which authoritative sources, under what conditions, with what permissions, in what order, for how long, and with what evidence?” These are architecture and governance questions.

> **Context Principle**
> The goal is not maximum context. The goal is the minimum sufficient, authoritative, timely, and safe context required for the decision or action.

2. Context Is Part of the System

An intelligent system is a sociotechnical system whose behavior emerges from the interaction of models, software, data, people, policies, and operating conditions. Context is the mechanism through which these elements become visible to the model or agent. It carries intent, constraints, facts, history, and authority into the reasoning process.

Because context shapes behavior, it should be treated as a first-class system component. Requirements should identify which decisions require which information. Architecture should define authoritative sources, retrieval paths, memory boundaries, and tool contracts. Security design should distinguish trusted instructions from untrusted content. Testing should evaluate behavior under correct, missing, stale, contradictory, and malicious context. Operations should observe what context was used when important actions occurred.

The model is not the system, and the context window is not the context architecture. The window is a limited execution surface. The architecture includes the upstream systems that collect, transform, select, rank, summarize, authorize, and deliver information into that surface. It also includes downstream mechanisms that preserve decisions, outcomes, and learning.

Context Element	Engineering Purpose
Intent and requirements	Define the outcome, constraints, acceptance conditions, and boundaries of action.
System and repository instructions	Provide durable engineering conventions, workflows, build rules, and validation expectations.
Authoritative domain data	Ground decisions in current facts, records, policies, and controlled knowledge.
Examples and demonstrations	Show acceptable reasoning patterns, outputs, and boundary cases.
Conversation and task state	Preserve continuity within a bounded interaction or workflow.
Long-term memory	Retain selected facts, preferences, decisions, and lessons beyond a single session.
Tool definitions and results	Expose capabilities, schemas, permissions, and the consequences of external actions.
Telemetry and operating signals	Provide evidence of system state, prior outcomes, and current risk.
Human input and approval	Inject judgment, authorization, correction, and accountability.

3. A Context Architecture for Intelligent Systems

Context architecture separates durable knowledge from transient execution state and separates trusted control information from untrusted content. Without these distinctions, systems accumulate a mixture of instructions, data, memories, and tool results that cannot be reliably governed.

A practical architecture contains several layers. The source layer includes repositories, databases, document systems, event streams, identity services, policies, and human input. The preparation layer validates, normalizes, classifies, labels, chunks, embeds, summarizes, and assigns provenance. The selection layer retrieves and ranks the information appropriate to the current task. The assembly layer constructs the actual context package, including precedence and token budgets. The execution layer invokes models and tools. The evidence layer records what was used, what happened, and why the result was accepted.

This layered model avoids a common error: allowing the model to discover and prioritize every source on its own. Intelligent systems need freedom to reason, but the organization must still define the search space, authority boundaries, and consequences. Context architecture is therefore a balance between flexibility and control.

Layer	Primary Responsibility	Key Evidence
Sources	Provide authoritative knowledge, policy, identity, state, and engineering information.	Source owner, classification, version, freshness, and access policy.
Preparation	Transform source material into usable and governable context.	Transformation logic, chunking, labels, summaries, and provenance.
Selection	Choose information relevant to the current task and risk.	Query, filters, ranking scores, selected sources, and exclusions.
Assembly	Construct the bounded context package and precedence order.	Instructions, token allocation, source ordering, and conflict resolution.
Execution	Reason, generate, invoke tools, and coordinate workflow.	Model, configuration, tool calls, approvals, and output.
Evidence and feedback	Record outcomes and improve the context environment.	Trace, evaluation result, incident finding, correction, and learning.

4. The Repository as Durable Engineering Context

For software engineering agents, the repository is one of the most important context environments. It contains source code, tests, architecture decisions, requirements, issue history, build definitions, release records, and operational evidence. A well-structured repository allows both humans and agents to understand what the system is, how it should change, and how a proposed change must be verified.

GitHub now supports repository-wide instructions, path-specific instructions, and agent instruction files such as AGENTS.md. The nearest instruction file in the directory tree can take precedence, allowing context to be scoped to the component being changed. [3] This is more than customization. It is a mechanism for expressing local engineering policy and preserving it close to the affected work.

OpenAI's experience with agent-first engineering reinforces the same principle: repository knowledge should be the system of record, but it should be navigable and layered. A single enormous instruction file becomes difficult to maintain and can overload the agent. Structured maps, component-level guidance, executable validation, and links to authoritative artifacts provide more useful context. [4]

Repository context should not duplicate everything an agent might need. It should point to authoritative sources and make important constraints discoverable. Requirements, ADRs, coding standards, test commands, dependency policies, data classifications, release expectations, and known limitations should be versioned and reviewable. The repository then becomes both the engineering record and an executable context surface.

Repository Principle
Documentation becomes operational when it changes what engineers and agents can correctly understand, decide, verify, and maintain.

5. Retrieval, Knowledge, and Grounding

Retrieval-augmented generation is one method of grounding model behavior in external knowledge. It can improve relevance and accuracy by searching proprietary or current information at request time. AWS Bedrock Knowledge Bases, for example, combine model inference with retrieval from organizational data sources. [5] Yet retrieval alone does not guarantee trustworthy context.

A retrieval system can return stale, incomplete, low-authority, duplicated, malicious, or semantically related but operationally irrelevant material. Engineering therefore requires source governance, metadata, filtering, access control, ranking, recency rules, and conflict resolution. It also requires an explicit answer to the question: what happens when the necessary context cannot be found?

Grounding should preserve provenance. The system should be able to identify the source, version, owner, retrieval query, ranking or selection basis, and time of use. NIST's AI Risk Management Framework emphasizes provenance and traceability as contributors to transparency and accountability. [6] The same logic applies to operational knowledge: an answer grounded in an obsolete policy is not trustworthy merely because it includes a citation.

Knowledge engineering also includes deciding what should not be retrieved. Sensitive information, privileged legal guidance, personal data, secrets, and restricted operational details may require special authorization or exclusion. Retrieval is therefore an access decision, not merely a search operation.

Grounding Question	Engineering Expectation
Authority	Is the source recognized as authoritative for this decision?
Freshness	Is the information current enough for the task and risk?
Relevance	Does the selected material directly support the current decision?
Completeness	Is critical context missing, truncated, or distributed across sources?
Conflict	How are contradictory sources detected and resolved?
Permission	Is the requester or agent allowed to access and use this information?
Provenance	Can the result be traced to the exact source and transformation path?
Fallback	Does the system abstain, escalate, or request clarification when grounding is insufficient?

6. Memory, State, and Continuity

Context exists across different time horizons. Short-term state supports a conversation or workflow. Episodic memory preserves selected prior experiences. Semantic memory retains durable facts or learned patterns. Procedural memory preserves methods, instructions, or skills. These forms should not be collapsed into one undifferentiated history.

Memory creates value by reducing repetition and supporting continuity, but it also creates risk. Incorrect memories can persist, sensitive information can be retained beyond its purpose, and a preference learned in

one context can be misapplied in another. Memory can become a hidden source of policy and behavior if it is not visible and governable.

OpenAI's Agents SDK guidance treats sessions as a mechanism for managing short-term memory and emphasizes pruning, compaction, and control over what remains in context. [7] Anthropic's work on long-running agents similarly uses compaction and external progress artifacts to avoid exhausting the context window while preserving task continuity. [8] Google research is exploring longer-term memory systems that distill successful and failed experiences into reusable reasoning strategies. [9] These developments show that memory is becoming an architectural subsystem rather than a simple transcript.

A professional memory design defines ownership, scope, retention, update rules, correction, deletion, access, and evidence. It also distinguishes remembered fact from inference. High-risk decisions should not depend on invisible or unverifiable memory.

Memory Type	Useful For	Primary Risk
Working state	Current task, conversation, plan, and tool results.	Context overflow, distraction, and accidental carryover.
Episodic memory	Prior cases, incidents, user interactions, and task outcomes.	Overgeneralization and retention of sensitive history.
Semantic memory	Stable domain facts, preferences, and organizational knowledge.	Staleness, false certainty, and unclear authority.
Procedural memory	Reusable workflows, agent skills, and engineering methods.	Unreviewed procedure changes and unsafe automation.
Reflective learning	Lessons distilled from successful and failed execution.	Self-reinforcing errors and unvalidated adaptation.

7. Tools, Protocols, and Interoperability

Context engineering increasingly depends on standard interfaces. The Model Context Protocol (MCP) defines an open protocol for connecting AI applications to external data sources and tools. It distinguishes resources, prompts, and tools and provides a common architecture for clients and servers. [10][11] MCP can reduce custom integration and make capabilities discoverable, but it does not remove the need for security, authorization, data classification, or tool-quality engineering.

Agent2Agent (A2A) addresses a related problem: communication and coordination between agents built by different vendors or frameworks. Google introduced A2A to support secure information exchange and coordinated action across enterprise systems, and later donated the project to the Linux Foundation for neutral governance. [12][13] MCP primarily helps an agent reach tools and context; A2A helps agents collaborate. Together, they signal a future in which context crosses organizational and platform boundaries.

Interoperability increases both capability and attack surface. Tool descriptions become executable context. Schemas influence what the model believes it can do. Agent cards, capability declarations, authentication, and delegation determine how work moves between systems. A context protocol should therefore be treated with the rigor of an API platform: versioning, compatibility, authorization, rate limits, auditing, testing, and lifecycle ownership.

The correct engineering posture is not to assume that a standard protocol is inherently safe. Standards make integration consistent; organizations remain responsible for deciding which servers, tools, resources, and agents are trusted for which purposes.

Protocol Principle
Interoperability standardizes the path through which context and authority move. It does not determine whether the context is trustworthy or the authority is appropriate.

8. Context Security and Trust Boundaries

Context is a major security boundary because models interpret data and instructions through the same reasoning surface. Untrusted content can attempt to override system intent, manipulate tool use, disclose sensitive information, or redirect the agent toward an attacker's objective. OWASP identifies prompt injection as a primary risk and notes that manipulation can be direct or embedded in external content. [14]

The security objective cannot be perfect detection of every malicious instruction. OpenAI's work on agent resistance to prompt injection argues for constraining the impact of manipulation through system design, risky-action controls, and protection of sensitive data. [15] This is consistent with traditional security engineering: assume some controls will fail and limit the consequence.

Trusted instructions, user requests, retrieved content, memory, and tool output should be labeled and handled differently. Systems should preserve instruction precedence, isolate high-risk data, validate tool inputs and outputs, use least privilege, and require confirmation for consequential actions. Authorization should be attached to identity and resource policy rather than inferred from natural-language context.

Context confidentiality also matters. Logs and traces may contain prompts, retrieved records, tool arguments, personal information, or secrets. Observability must therefore support redaction, selective capture, access controls, and retention policies. The most complete trace is not always the safest trace.

Threat	Context-Engineering Response
Prompt injection	Separate trusted instructions from untrusted content; constrain actions even if manipulation succeeds.
Sensitive information disclosure	Classify sources, enforce access controls, redact telemetry, and minimize retained context.
Context poisoning	Validate source ownership, integrity, freshness, and change provenance.
Tool manipulation	Use typed schemas, allowlists, validation, least privilege, and approval for consequential actions.
Memory corruption	Require controlled writes, correction, expiry, and auditability for durable memory.
Cross-agent trust abuse	Authenticate agents, verify delegated authority, and constrain shared context.
Context overflow and distraction	Budget, prioritize, summarize, and test for lost or contradictory instructions.

9. Evaluation, Observability, and Context Quality

A context architecture cannot be trusted because its sources appear reasonable. It must be evaluated. Context evaluation asks whether the system selected the right information, excluded unsafe or irrelevant material, preserved critical instructions, resolved conflicts, and produced behavior consistent with requirements.

Agent evaluations should separate model capability from context quality. A failure may result from a poor model, an incomplete tool description, a missing document, stale memory, a retrieval ranking error, or an

ambiguous policy. Anthropic emphasizes that evaluations make behavioral changes visible before they reach users and that their value compounds across the agent lifecycle. [16]

Observability provides the evidence needed to diagnose these failures. OpenTelemetry's generative AI conventions are evolving to capture model requests, token usage, retrieval, tool calls, tool results, and agent operations through standardized traces, metrics, and events. [17][18] An effective trace should connect the user or system intent to the assembled context, model execution, tool use, approvals, output, and outcome.

Context-quality metrics should not become vanity measures. Retrieval precision, token utilization, cache hit rate, and latency are useful, but they do not prove decision quality. The strongest metrics connect context behavior to outcomes: fewer unsupported claims, safer tool use, lower escalation due to missing information, improved defect detection, or reduced incident recurrence.

Evaluation Dimension	Example Evidence
Sufficiency	The context contained the minimum information needed to complete the task correctly.
Authority	Selected sources were appropriate and traceable for the decision.
Freshness	The system used current versions of policies, code, data, and instructions.
Conflict handling	Contradictions were detected, surfaced, or resolved according to policy.
Security	Untrusted content did not override authority or expose restricted information.
Efficiency	Context was bounded, prioritized, and cost-effective without losing critical information.
Outcome quality	The resulting decision or action satisfied requirements and avoided unacceptable harm.
Reproducibility	The organization can reconstruct the important context behind a consequential action.

10. Context Lifecycle and Governance

Context has a lifecycle. It is created, reviewed, published, selected, transformed, consumed, observed, corrected, deprecated, and retired. Organizations often govern source code and production data while leaving prompts, retrieval configurations, memory rules, tool descriptions, and agent instructions comparatively informal. That gap will become increasingly dangerous as agents gain authority.

Governance should define owners for context sources and assembly logic. Changes to high-impact instructions, policy mappings, retrieval filters, tool schemas, or memory behavior should be reviewed and versioned. Release evidence should identify the context configuration associated with the deployed system. Incidents should preserve enough information to reconstruct which context influenced the behavior.

NIST AI RMF and its Generative AI Profile emphasize continual risk management, information integrity, human-AI configuration, privacy, and transparency. [6][19] These outcomes require more than model documentation. They require governance over the information environment in which the model operates.

Context governance should be proportionate. A low-risk writing assistant does not need the same evidence as an agent that changes financial records or operates infrastructure. The governing principle is consequence: the greater the authority and potential impact, the stronger the requirements for source control, provenance, evaluation, approval, observability, and intervention.

Lifecycle Activity	Governance Question
Create	Who may author instructions, tools, memories, policies, and knowledge sources?
Review	What expertise and independence are required before use?
Publish	How is the approved context identified, versioned, and made discoverable?
Select	Which identity, task, risk, and permission rules control access?
Transform	How are summaries, embeddings, labels, and derived context validated?
Use	What evidence records the context used for a consequential action?
Correct	How can inaccurate or harmful context be amended, invalidated, or removed?
Retire	How are obsolete sources and memories prevented from continuing to influence behavior?

11. Context for AI-Assisted Software Engineering

Software engineering is an especially demanding context problem. A coding agent must understand the repository, issue intent, architecture, local conventions, dependencies, tests, build environment, security requirements, release rules, and current state of the work. Generic model knowledge is not enough.

GitHub recommends repository instructions that explain how to understand the project and how to build, test, and validate changes. [3][20] Agent skills can package instructions, scripts, and resources for specialized work. [21] These mechanisms turn previously informal team knowledge into reusable context. They also expose whether the team's engineering process is explicit enough for another engineer—or an agent—to follow.

Context engineering does not mean documenting every detail. It means preserving the high-value information that prevents incorrect assumptions and enables verification. A good issue communicates intent and acceptance criteria. An ADR explains the relevant architectural decision. A test command makes validation executable. A runbook explains operational response. A release record identifies the approved baseline. Together, these artifacts create a context chain.

AI-assisted development therefore rewards repository quality. Weak repositories force agents to infer architecture, standards, and procedures from scattered code and history. Strong repositories provide navigable evidence and explicit control points. This is why repository-centered engineering and context engineering are becoming mutually reinforcing disciplines.

> **Engineering Principle**
> The quality of an engineering agent is bounded not only by the model, but by the quality of the engineering environment the organization has made legible to it.

12. Failure Modes and Context Theater

Context engineering can become theater when teams accumulate documents, embeddings, memory, and instructions without demonstrating that these elements improve decisions. More sources, larger windows, and richer memory may create confidence while making behavior less predictable.

Failure Mode	What It Looks Like	Consequence
Context dumping	Large volumes of documents are inserted without ranking, precedence, or purpose.	Critical constraints are diluted and cost increases.
Stale authority	Obsolete policies, code, or procedures remain retrievable.	The system confidently applies superseded rules.
Unbounded memory	The system retains history without scope, correction, or deletion.	Privacy, bias, and behavioral drift accumulate.
Instruction conflict	Multiple prompts and policy sources disagree silently.	Behavior changes by ordering or implementation detail.
Tool description as policy	Natural-language tool metadata is treated as sufficient authorization.	Agents perform actions beyond intended authority.
Retrieval without provenance	Sources are returned without version, ownership, or traceability.	Results cannot be defended or reproduced.
Observability without privacy	Full context is logged indiscriminately.	Sensitive data and secrets spread into telemetry.
Protocol optimism	MCP or A2A integration is assumed safe because it is standardized.	Interoperability expands trust beyond governance.
Context metrics theater	Teams optimize tokens and retrieval scores without measuring outcomes.	Efficiency improves while decision quality remains unknown.

13. A Context Engineering Operating Model

Context engineering crosses product, software, data, security, governance, and operations. It cannot be assigned solely to prompt authors or AI specialists. Organizations need a federated operating model in which platform teams provide shared capabilities while domain teams own the meaning and consequences of their context.

A context platform may provide source connectors, retrieval services, memory stores, protocol gateways, policy enforcement, observability, evaluation infrastructure, redaction, and cost controls. Domain teams determine authoritative knowledge, risk, workflow, and acceptable behavior. Security teams define trust boundaries and authorization. Engineering teams integrate context into architecture and release processes. Operators monitor runtime behavior and feed incidents back into context design.

The repository should preserve the context design: source inventories, instruction hierarchy, memory policy, tool contracts, evaluation sets, context-risk decisions, release configuration, and incidents. This creates continuity as models and platforms change.

Operating-Model Element	Design Question
Context owner	Who is accountable for the quality and lifecycle of each important context source?
Context architect	Who designs source, preparation, selection, assembly, memory, and evidence patterns?
Platform capability	Which shared services provide retrieval, memory, protocols, security, telemetry, and evaluation?
Domain stewardship	Who decides which facts, rules, examples, and outcomes are authoritative?
Security and privacy	Who defines trust boundaries, access, retention, redaction, and incident controls?

Operating-Model Element	Design Question
Release governance	How are context changes reviewed, versioned, tested, and associated with a release baseline?
Operational learning	How do failures, corrections, and outcomes improve sources, memory, and evaluation?
Executive oversight	Which risks, outcomes, and investments require organizational decision and accountability?

14. A Five-Level Context Engineering Maturity Model

Maturity Level	Characteristics
Level 1 — Ad Hoc	Prompts and documents are assembled manually. Sources, versions, and authority are unclear. Failures are attributed to the model.
Level 2 — Documented	Teams maintain prompts, retrieval sources, and basic instructions, but governance and evaluation are inconsistent.
Level 3 — Engineered	Context architecture, source ownership, memory policy, versioning, evaluation, and repository guidance are defined and repeatable.
Level 4 — Governed	Risk-tiered controls, provenance, authorization, release baselines, security testing, and observability govern consequential context use.
Level 5 — Adaptive and Evidence-Driven	Context is continuously improved from measured outcomes, incidents, and evaluation while preserving human accountability and controlled change.

Maturity does not mean maximizing automation. A Level 5 organization may deliberately require human confirmation or restrict memory and tool access in high-risk workflows. Maturity means that those decisions are explicit, evidence-based, and sustainable.

15. The ETIS Position

ETIS treats context as a control surface that spans the entire engineering lifecycle. Context begins with intent and requirements, becomes structured through architecture and repository evidence, guides implementation and review, accompanies release, and is observed in operation. It includes not only information supplied to models, but also the policies, identities, permissions, tools, state, memory, and human decisions that bound intelligent behavior.

This view leads to several conclusions. Context should be authoritative rather than merely available. It should be minimal and task-relevant rather than maximal. It should preserve provenance and lifecycle ownership. It should distinguish trusted control information from untrusted content. It should be evaluated under normal, missing, conflicting, stale, and adversarial conditions. Its consequential use should be observable and reconstructable.

Repository-centered engineering gives context a durable home. Evidence-centered engineering makes its effects reviewable. Engineering governance defines who may change it. Review and readiness determine whether it is fit for release. Operational readiness provides intervention and learning. Agentic engineering increases the stakes because context can now influence action at machine speed.

ETIS Position

Context is architecture. Context is governance. Context is evidence. A trustworthy intelligent system requires a context environment that is intentionally designed, bounded, versioned, evaluated, observable, and owned.

Implications for Leaders, Engineers, and Educators

For executives, context engineering is an enterprise capability. It affects whether organizational knowledge can be safely mobilized, whether agents can operate across systems, and whether decisions can be explained. Investment should therefore extend beyond model access to knowledge stewardship, identity, protocols, evaluation, observability, and operating-model ownership.

For engineers, context engineering expands the definition of system design. Requirements, documentation, tool schemas, memory, retrieval, and telemetry are no longer supporting concerns. They are part of the behavior-producing architecture. Engineers must be able to reason about information authority, trust boundaries, temporal validity, and the consequences of context failure.

For educators, context engineering provides a practical reason to teach requirements, architecture, repositories, data, security, testing, and operations as an integrated discipline. Students who use AI tools should learn to improve the engineering environment around the tool, not merely write better prompts. They should be able to explain what context was provided, why it was trusted, how it was verified, and what happened when it was incomplete or wrong.

Conclusion

The next generation of intelligent systems will not be differentiated by model capability alone. Models will increasingly be interchangeable, routed, specialized, and embedded in larger systems. The enduring engineering advantage will come from the quality of the environment in which those models reason and act.

Context engineering is the discipline for building that environment. It connects organizational intent to execution, repositories to agents, knowledge to decisions, memory to continuity, protocols to interoperability, and telemetry to accountability. It transforms context from an informal collection of prompts and documents into an engineered system with architecture, ownership, controls, evidence, and lifecycle.

As agents gain greater authority, context quality becomes operational risk. A trustworthy system must know not only how to reason, but what it is allowed to know, which sources it should trust, what it may do, when it must ask, and how its decisions can be reconstructed. Context engineering is therefore not a temporary technique around current models. It is becoming a durable branch of software engineering for the intelligent-systems era.

Final Takeaway
Models provide capability. Context gives that capability direction, boundaries, continuity, and meaning. Engineering the context is how organizations turn model potential into trustworthy system behavior.

References

[1] Anthropic. (2025). Effective context engineering for AI agents. https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

[2] Amazon Web Services. (2025). Generative AI lifecycle operational excellence framework. https://docs.aws.amazon.com/prescriptive-guidance/latest/gen-ai-lifecycle-operational-excellence/

[3] GitHub. (2026). Adding repository custom instructions for GitHub Copilot. https://docs.github.com/copilot/customizing-copilot/adding-custom-instructions-for-github-copilot

[4] OpenAI. (2026). Harness engineering: Leveraging Codex in an agent-first world. https://openai.com/index/harness-engineering/

[5] Amazon Web Services. (2026). Knowledge bases for Amazon Bedrock. https://docs.aws.amazon.com/bedrock/latest/userguide/knowledge-base.html

[6] National Institute of Standards and Technology. (2023). Artificial Intelligence Risk Management Framework (AI RMF 1.0). https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf

[7] OpenAI. (2025). Short-term memory management with sessions. https://developers.openai.com/cookbook/examples/agents_sdk/session_memory

[8] Anthropic. (2025). Effective harnesses for long-running agents. https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents

[9] Google Research. (2026). ReasoningBank: Enabling agents to learn from experience. https://research.google/blog/reasoningbank-enabling-agents-to-learn-from-experience/

[10] Model Context Protocol. (2025). Introduction. https://modelcontextprotocol.io/docs/getting-started/intro

[11] Model Context Protocol. (2025). Specification 2025-11-25. https://modelcontextprotocol.io/specification/2025-11-25

[12] Google Developers. (2025). Announcing the Agent2Agent Protocol (A2A). https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

[13] Google Developers. (2025). Google Cloud donates A2A to the Linux Foundation. https://developers.googleblog.com/en/google-cloud-donates-a2a-to-linux-foundation/

[14] OWASP GenAI Security Project. (2025). LLM01: Prompt injection. https://genai.owasp.org/llmrisk/llm01-prompt-injection/

[15] OpenAI. (2026). Designing AI agents to resist prompt injection. https://openai.com/index/designing-agents-to-resist-prompt-injection/

[16] Anthropic. (2026). Demystifying evaluations for AI agents. https://www.anthropic.com/engineering/demystifying-evals-for-ai-agents

[17] OpenTelemetry. (2024). OpenTelemetry for Generative AI. https://opentelemetry.io/blog/2024/otel-generative-ai/

[18] OpenTelemetry. (2026). Inside the LLM call: GenAI observability with OpenTelemetry. https://opentelemetry.io/blog/2026/genai-observability/

[19] National Institute of Standards and Technology. (2024). AI RMF: Generative Artificial Intelligence Profile. https://www.nist.gov/publications/artificial-intelligence-risk-management-framework-generative-artificial-intelligence

[20] GitHub. (2026). Best practices for using Copilot coding agent to work on tasks. https://docs.github.com/copilot/tutorials/cloud-agent/get-the-best-results

[21] GitHub. (2026). Adding agent skills for GitHub Copilot. https://docs.github.com/copilot/how-tos/copilot-on-github/customize-copilot/customize-cloud-agent/add-skills

[22] OpenAI. (2026). Context engineering for personalization. https://developers.openai.com/cookbook/examples/agents_sdk/context_personalization

[23] Anthropic. (2025). Writing effective tools for AI agents. https://www.anthropic.com/engineering/writing-tools-for-agents

[24] Google Research. (2026). Towards a science of scaling agent systems. https://research.google/blog/towards-a-science-of-scaling-agent-systems-when-and-why-agent-systems-work/

[25] OWASP GenAI Security Project. (2025). Top 10 for Agentic Applications for 2026. https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/

[26] Amazon Web Services. (2026). Agentic AI frameworks, platforms, protocols, and tools on AWS. https://docs.aws.amazon.com/prescriptive-guidance/latest/agentic-ai-frameworks/introduction.html

[27] Microsoft. (2026). Microsoft Agent Framework overview. https://learn.microsoft.com/en-us/agent-framework/overview/

[28] SLSA. (2026). Provenance. https://slsa.dev/provenance

[29] OpenTelemetry. (2025). AI agent observability: Evolving standards and best practices. https://opentelemetry.io/blog/2025/ai-agent-observability/

[30] Anthropic. (2024). Building effective agents. https://www.anthropic.com/engineering/building-effective-agents

About the Author

William T. O'Connell, Ph.D. is an Adjunct Professor of Computer Science at Loyola University Chicago and the creator of the Engineering Trustworthy Intelligent Systems (ETIS) framework. His professional career spans more than four decades of software engineering, architecture, delivery, quality, and technology leadership, including service as an IBM Distinguished Engineer and CTO for IBM Consulting Quality and Delivery. His current work focuses on repository-centered engineering, engineering evidence, trustworthy intelligent systems, agentic software delivery, and the preparation of students and practicing engineers for increasingly AI-assisted engineering environments.

This white paper is part of the ETIS White Paper Series, which examines the engineering practices required to build, govern, review, operate, and sustain trustworthy software and intelligent systems.

Suggested citation: O'Connell, W. T. (2026). Context Engineering: Designing the Information, Memory, and Control Environment for Intelligent Systems. ETIS White Paper Series, WP-009, Version 1.0.