Context Engineering within Prompt Auto-Generation

As GenAI emerges as an active collaborator in business, such as healthcare, engineering, writing, and so forth, it provides decision support for humans in the loop, or on the loop monitoring its output. In terms of auditability, AI output becomes a legal artifact that requires traceability when it crosses the threshold of influencing a financial, medical, legal, or operational decision point: Each of these domains are highly regulated. The expectation from adjacent control regimes does not care about how the output was generated. Rather whether the organization in question can stand behind the output, justify it to regulators.

Reproducibility

More intelligence only serves to increase liability exposure unless accompanied by 'reproducibility' and stronger evidence of traceability. Reproducibility is a systems engineering problem at its heart. When the work isn't executable outside a narrow, fragile environment, it is not a valid artifact. And so, to that end, the manner in which the output was generated at a point in time must be documented. The organization's procedures depend on scope, requirements, and constraints that are a process whether executed by humans or machine code. Success criteria cannot be implicit. Regulators do not accept 'probability' and 'situational understanding' as a defense. Instead, the AI engineer must show that a repeatable procedure produces evidence that is grounded in reality; that the system can inspect and assess observable artifacts. What constraints were in place at the time of the query are also called into account. All of this leads to the need for controlling context via engineered algorithms.

The emerging concern of reproducibility central to Generative AI, indeed any AI interaction, addresses the requirement to provide context to the system, model, or architecture. Indeed, context management is rapidly becoming a central architectural concern of any agentic framework seeking to query LLMs, foundation models, or pre-trained subsystems managing industrial information at scale.

Failure states

LLMS and, as a result, Agents fail due to vague or imprecise instructions, taking the wrong action or failing to do what the user expected. Here are a few reasons for this phenomenon:

1. The LLM is not capable enough to accomplish the request: one-shot, unable to handle edge cases, inflexible, lack of iteration. (context structure)
2. The context is not passed to the LLM in the prompt in the right way or at all: ambiguous, irrelevant, contradictory, or vague. (context confusion)
3. The LLM gets bogged down with too much text and loses focus because it is overloaded. (context distraction)

Failure to constrain reasoning results in passive, canned responses rather than active collaboration on the part of the token-driven subsystem. Done right, Context Engineering provides the LLM the right information and tools, versioned data, in the correct format, to accomplish the task at hand. Most often, the AI Prompt Engineer is defining an end goal, instead of considering the process the LLM goes through. First, the model call through an API, then orchestrating the context, information, and tooling necessary to reach a desired output. It is a process like any other, hence the term “engineering.”

Context

Creating context is tricky for even the most experienced engineers. You are imposing a world view on the model and the settings from which to act. That effort requires understanding that models are purpose built and designed for a particular task: coding, information retrieval, and so forth. While some are general purpose, when working with any model, understanding the background and domain of any information agent requires building a prompt that takes into consideration a variety of contexts: Model context, tooling context, business goals and desired outcomes.

When you explore how models are built a troubling pattern emerges. When a company uses a book to develop an AI model, it splits the book’s text into tokens or word fragments. For example, the phrase *Mr. & Mrs. Dursley, of number four, Privet Drive* might be represented by the tokens *Mr.*, *&*, *Mrs.*, *Dursley*, and so forth. Some tokens are actual words; some are just groups of letters, spaces, and punctuation. The model stores these tokens *and the contexts* in which they appear in files. The resulting LLM is essentially a huge database of contexts and the tokens that are most likely to appear next in a basic decision tree of probabilities. It rapidly steps through the choices within a forest of data, making each choice based on a series of training data. Note that these models are not trained on ‘human language’ per se, but on selected books, articles, research, and other text sources scraped from the internet. Models have been

proven to not only reproduce portions of works word for word but also that a model's output can paraphrase a book rather than duplicate it exactly.

Drift

A key issue with models is that they drift (because of those statistical, probabilistic branches), and have a built-in bias resulting from the selection of training data used to create them. With transient (a single call) as well as persistent context (state data saved across calls) baked into the structure and predictable "next word" methodology of results, the prompt bears the burden of narrowing the task to a specific set of requirements. Most prompt engineers fail to dissect the atomic steps of the problem based on a targeted goal, and rather test and refine through multiple real-time calls (trial and error) to get to that goal. Without a roadmap, they waste a lot of tokens and time tuning and adjusting their requests to the model.

Requirements & tooling

Like any good software design, prompt engineering requires a specification. Steps, processes, data sources, state machines, tools, output format—all these and more create a type of "middleware" that mediates between the model and operator while tracking the lifecycle of the agent and project.

Model	The actual model and version to be called, including any configuration data
Tools	The set of utilities or packages the process can leverage to take actions in advance of the query or post-processing
Response format	Schema or specification for the final response
Messages	Conversation history with the LLM, stored or pulled from short-term memory and stored in a local database, log, or other auditable location

Only after these precursor elements are taken into consideration, can a base set of instructions be crafted into a system prompt from the engineer and sent to the model, generally through an API call.

Typically, experts speak of the ***4 S-es*** of prompt engineering: specificity (precise instructions), simplicity (concise prompts), structure (logical format), and strategy (best method for the task). From this perspective, it's less 'engineering' in the traditional sense and more correctly typified as mastering the art of conversations with advanced systems (human to machine). In other words, how to tell the LLM a story—a back and forth that belies the idea of 'one-shot' prompts.

So goes the story: in order to interrogate an LLM, one must have a strategy and prep in advance, just like a moderator before a debate. This is the first S. Without a strategy, you're going in blind. From strategy to structure the process of thinking through your interactions may involve a step by step roadmap, a series of agents managing other agents via programmatic tool calling, or a template for how to proceed.

Finally, you have the necessary ingredients to write precise, concise instructions. Logical and well defined, the prompt takes on the nature of a mini-program, perhaps a python or java script that can be fed through an API, debugged like any other code, and refined logically instead of haphazardly. Trial and error moves to measurable, sustainable, controlled work product. You end up with a conversational session that is efficient, reliable, repeatable, and in a sense friendly.

Strategy	How are you approaching the situation, or business goal? Is there a particular tone or style to adopt? What persona is acting as 'narrator' to your story? Provide a focus.
Structure	Will you use templates, a process, iterate with multiple agents? Some recommend 'chain of thought' or 'persona-driven' prompts.
Simplicity	Target the context to the business goal; limit speculative or risky information; avoid data privacy areas; avoid cliches
Specificity	Avoid ambiguity; be logical; provide examples; be factual; provide step-by-step tasks

Structured context is critical to designing better prompts. But what provides structure and intelligence that leads to better results from LLMs? When the model does not understand the internal knowledge that drives your business, there is no grounding in reality and facts. This is where ontologies and data dictionaries come into play. By garnering metadata about the domain, topic, or theme of your system to augment the prompt, you in essence are providing a 'shared vocabulary' between humans and machines. Metadata examples include domain, component, genre, topic, dependencies, constraints and so forth.

By providing robust context at the system prompt level, when following up with the user prompt, the LLM has a pre-existing set of conditions that act as boundaries and limitations on its response. This approach reduces hallucinations and eliminates vague, generic responses. Recommendations and analysis become grounded in a set of confidence parameters and are less often just plain wrong.

Can Ontologies Help?

Ontologies and their close cousin data dictionaries, act as anchors for the LLM. Thinking through the parameters and constraints of the problem being researched, or even just determining the domain-specific way of talking about things, gives your conversation with the machine a more human, real-world set of terms, facts, and concepts. It is a type of ubiquitous language that can then be standardized, reused for each area of expertise. The common language is a powerful goal that ontologists and developers have long sought to establish. The problem is that no one agrees on what that common language and its structure should look like.

With ontologically-driven prompt design the question of 'common' terms and agreed-upon structures are eliminated as each user can customize to their heart's content and the LLM will respond in kind. The user reduces ambiguity and improves communications with the model on their own terms. Literally. This approach grounds the model with clearer, reusable language and solves the 'memory' problem.

That 'memory' problem is based on research showing that models 'forget' (or experience *attention decay*) about the history of prompting after about 8 iterations.[1] And there is not yet a clear way to force the model to increase its range of refinement in prompt-based conversations. Instead, taking an older technical approach of storing the context in another modality, then referencing it as needed pre-injects context into the prompt at the very beginning making the subsequent task-oriented prompts more efficient.

In the same vein, you can specify the desired output by referencing templates, formats, and constraints that are stored in data dictionaries. Consider the medical field where elements such as compliance, record type, and criticality factors are common. Submitting and resubmitting claims often boils down to a formulaic expectation on the part of the insurance company. Doctors using metadata about a specific insurer's preferred vocabulary can craft better documentation with AI and save hours of administrative overhead.

Here are some ways to accomplish prompt crafting using metadata from ontologies and data dictionaries:

- Use MCP (Model Context Protocol) for standardizing metadata inputs
- API specs that provide structured prompt inputs
- Create a metadata block at the start of every prompt; label it as a 'system prompt' so that it maintains control over everything that follows.

[1] "Testing popular models like LLaMA2-chat-70B, we reveal a significant persona drift within eight rounds of conversations. An empirical and theoretical analysis of this phenomenon suggests the transformer attention mechanism plays a role, due to *attention decay* over long exchanges." *Measuring and Controlling Persona Drift in Language Model Dialogs* (arXiv:2402.10962)

- Put your ontology or data dictionary into JSON or YAML so that there is a shared programmatic consistency.
- Quality check your prompts with validation scripts to check for missing context before sending the prompt through the API; this saves tokens and therefore money.

JSON/YAML ontologies define *what* data exists, however they don't address ambiguity **at** the tool definition layer itself. Schemas define structure but can't express:

- When to include optional parameters
- Format conventions (date formats, ID patterns)
- Which parameter combinations make sense
- API-specific conventions

Grounding prompts in reality results in better outcomes. Programmatically following a specification related to your business and its goals, testing the templated prompt, then reiterating after evaluating the response will result in a more automated and therefore repeatable and measurable set of interactions with an LLM. Know yourself, know your business, know your model. It’s as simple, and as complex, as that.

Tool Orchestration—A Way Forward

Approaches such as Programmatic vs Natural Language create opportunities for innovation and creativity in mapping out a path to measured improvements:

- **Parallel execution** (use the Python function asyncio.gather instead of sequential inference)
- **Result filtering** (only final output returns to context, not intermediate data)
- **Explicit control flow** (loops, conditionals, transformations in code)
- **Measured improvements:** 37% token reduction + reduced latency + improved accuracy

Traditional (Natural Language Tool Calling):

- Claude calls tool → receives result → reasons → calls next tool
- Each inference pass: 500ms-2s
- For 5 calls: ~2.5s latency + 5x result accumulation in context
- Intermediate data pollutes context window

Programmatic (Code-Based Orchestration):

- Claude writes orchestration code once (Python)

- Code executes tools in parallel or sequence
- Only final output returns to Claude's context
- For 5 parallel calls: same result, no latency overhead, no context pollution

Use Programmatic Tool Calling when:

- Multi-step workflows with 3+ tool calls
- Large intermediate results need filtering/transformation
- Parallel operations (checking 50 endpoints)
- Loops or conditionals needed to orchestrate

Dynamic Tool Discovery

Not all tools are always available in context: With 50+ MCP tools loaded upfront, you consume 55K-134K tokens before any work begins.

Anthropic's solution: Tool Search Tool

Implements deferred loading:

- Mark tools with defer_loading: true
- Claude searches for needed tools on-demand
- Only 3-5 relevant tools loaded per task
- Preserves 95% of context window

Layered Context Loading Strategy

1. Always Loaded (System Prompt + Core Context):
- Business rules and domain ontology
- Tool Search Tool (~500 tokens)
- 3-5 most-used tools (frequently needed)

2. Search-Discovered (On-Demand):
- All remaining tools marked with defer_loading: true
- Discovered when Claude searches for specific capability
- Automatically expanded into full definition

3. Orchestration Context (Code Execution):
- Tool definitions converted to Python functions
- Executed in sandboxed environment

- Results processed outside Claude's context window

These approaches incur a benefit of access to the full tool library with 95% context preservation.

Advanced Tool Patterns: Discovery, Invocation, Orchestration

As AI systems scale to hundreds of tools, three architectural patterns emerge:

1. Tool Search (Dynamic Discovery)
   When tool definitions exceed 10K tokens or accuracy drops with large tool sets, use Tool Search for on-demand discovery. Example: 50+ MCP tools across GitHub, Slack, Jira, Sentry, Grafana = 55K tokens → 8.7K with search (85% reduction).
2. Tool Use Examples (Precise Invocation)
   When ambiguity exists in optional parameters, format conventions, nested structures, or similar tools, provide concrete examples. Results: 72% → 90% accuracy on complex parameters.
3. Programmatic Tool Calling (Efficient Orchestration)
   When workflows require 3+ dependent tool calls, large result filtering, parallel execution, or complex control flow, use code-based orchestration. Results: 37% token reduction, reduced latency, improved accuracy.

Best Practice: Layer Strategically
Don't use all three for every task. Start with your biggest bottleneck:

- Context bloat from definitions? → Start with Tool Search
- Intermediate results polluting context? → Start with Programmatic Calling
- Parameter errors? → Start with Tool Use Examples

Then layer additional features as needed.

Integrated Architecture: Three Complementary Layers

Your Context Engineering strategy should span three layers that work together:

1. Tool Discovery (Tool Search Tool)
   Problem: Too many tool definitions consume context
   Solution: Search-based on-demand loading
   Benefit: 85% context preservation
   Source: Anthropic testing with 50+ MCP tools
2. Tool Invocation (Tool Use Examples)
   Problem: Ambiguity in parameter usage, wrong tool selection
   Solution: Concrete examples of correct usage patterns

Benefit: 72% → 90% accuracy on complex parameters
Source: Anthropic internal testing

3. Tool Orchestration (Programmatic Tool Calling)
   Problem: Intermediate results pollute context, multiple inference passes
   Solution: Code-based orchestration with result filtering
   Benefit: 37% token reduction + reduced latency + improved accuracy
   Source: Anthropic testing on complex research tasks

Analysis: These layers are complementary: discover the right tools → invoke them correctly → execute efficiently.

Enhanced Implementation Roadmap

Your document mentions implementation approaches but lacks a sequenced plan. Here's what to add:

Phase 1: Foundation (Weeks 1-2)

- Define domain ontology in JSON/YAML 1,2
- Identify business goals and core tools
- Mark most-used tools for always-loaded (defer_loading: false)

Phase 2: Tool Specifications (Weeks 2-3)

- Add Tool Use Examples to ambiguous tools
- Document return formats for orchestration code
- Enable Tool Search for library of 10+ tools

Phase 3: Orchestration (Weeks 3-4)

- Identify multi-step workflows (3+ tool calls)
- Build orchestration code templates using Programmatic Tool Calling
- Test token savings and accuracy improvements

Phase 4: Validation (Weeks 4-5)

- Measure improvements: token efficiency, accuracy, latency
- Document learnings and refine
- Implement governance for ontology maintenance

Conclusion

Structured context reduces hallucinations and eliminates vague, generic responses. And research at Anthropic validates this. All three advanced tool features prove this principle works at scale:

- Tool Use Examples improve accuracy from 72% → 90%
- Programmatic Tool Calling improves knowledge retrieval from 25.6% → 28.5%
- Tool Search allows precision with 50+ tools instead of context bloat

Context Engineering works because it grounds LLM reasoning in structured, domain-specific knowledge. Recent advances in tool search, usage examples, and programmatic orchestration extend this principle to systems with hundreds of tools. By layering discovery, invocation, and orchestration patterns, teams can build AI agents that operate with precision, efficiency, and measurable accuracy improvements.