Semantic AI Framework for Prompt Engineering

Dmitry Lande[1], Leonard Strashnoy[2]
[1] ORCID: 0000-0003-3945-1178
National Technical University of Ukraine – Igor Sikorsky Kyiv Polytechnic Institute
[2] ORCID: 0009-0008-5575-0286
University of California, Los Angeles (UCLA)

Annotation – The article presents a semantic framework for prompt engineering that utilizes natural language for creating and managing artificial intelligence systems. The primary goal is to represent prompts as analogs of programming constructs (conditional statements, loops, functions) through the mathematical formalization of their logic and interactions. The paper describes key primitives ("building blocks"), such as "Condition," "Loop," and "Function," along with methods for composing these primitives to construct complex systems, including semantic networks. Optimization techniques for prompts are explored, including gradient descent in the instruction space and a "swarm of virtual experts" model to enhance accuracy. Additionally, the concept of a "prompt language" is proposed as a domain-specific language for no-code AI development, lowering the entry barrier for users and facilitating the creation of complex logical structures through textual instructions.

Keywords:
prompt engineering, no-code approach, mathematical formalization, semantic network, primitives, composition, swarm of virtual experts, prompt language

1. Introduction

The development of large language models (LLMs) is shaping a new reality in which traditional programming is giving way to a different approach to problem-solving. In the past, programmers created algorithms, wrote code, and designed databases. Today, these processes are increasingly delegated to artificial intelligence systems. The interface for interacting with data is shifting from complex coding to natural language, where the primary tool becomes prompts — text-based queries that define the logic of model operations.

This transformation echoes the ideas of Stephen Wolfram, expressed in his book A New Kind of Science [Wolfram, 2002]. Wolfram demonstrated how complex natural phenomena, such as diffusion or pattern formation, can be modeled using simple cellular automata without requiring complex equations. His approach showed that basic rules are capable of generating extraordinarily intricate behavior. Although some aspects of this concept remain debated, the analogy with cellular automata is well-suited for describing the new paradigm of working with data, where language models serve as the core of information processes.

LLMs function as systems that do not require predefined rigid rules or algorithms. Instead, they adapt to given tasks based on the context provided through prompts. As a result, modern analysts and developers can solve problems that previously required months of work in just minutes. For example, rather than writing a complex SQL query to identify patterns in a dataset, a prompt engineer can simply ask the model in natural language: *"What are the most significant correlations in this data?"*

However, as with cellular automata, this approach is neither absolute nor all-encompassing. Its effectiveness depends on numerous factors, including data quality, the clarity of prompt formulation, and the model's ability to generalize. This suggests that the future will not be entirely devoid of technical specialists, but their numbers will significantly decrease. Programmers will remain essential at lower levels—for hardware optimization, the development of new model architectures, or improvements to training algorithms. Meanwhile, most other tasks will be handled through a combination of prompts, forming a kind of *"swarm of virtual experts"*—a set of queries operating simultaneously and interacting to achieve optimal results.

The future of programming resembles an evolutionary process. Just as Wolfram's cellular automata demonstrated that simple rules can generate complex structures, today's LLMs are opening new frontiers in information management. However, as with cellular

automata theory, this approach has its limitations. It will not eliminate all challenges but will fundamentally reshape the role of humans in the creation of information systems. The number of specialists will decrease, but they will focus on higher-level tasks, leaving routine operations to artificial intelligence.

In this new reality, the key skills are no longer knowledge of programming language syntax but rather the ability to formulate tasks in a way that language models can understand and execute. The profession of prompt engineering is becoming analogous to that of an architect — designing systems not with code but with a set of rules, queries, and linguistic structures. Strong prompt engineers are likely to be individuals who can think analytically, formulate clear and precise queries, and consider the nuances of human language. These could include linguists, methodologists, and data analysts accustomed to working with abstractions. Interdisciplinary thinking skills, which allow for the consideration of various contexts, will become highly valuable.

A critical question arises: can models themselves generate prompts, eliminating the need for human involvement? While models already demonstrate the ability to generate effective queries, fully excluding humans from the process is unlikely. The fundamental issue is that even the most advanced models cannot independently define ultimate goals or assess the broader implications of their actions. Humans remain essential for setting high-level objectives, evaluating results, and making ethical and strategic adjustments. The interaction between humans and models is synergistic — models enhance human capabilities, but the final decision remains with people.

The transition to no-code systems and the shift in the role of specialists toward working with LLMs signifies a new era of technological advancement, where interaction with models becomes the primary tool for innovation. Prompt engineering is already shaping a new profession that combines analytical skills, creativity, and technological understanding. However, human involvement does not disappear — it evolves. Humans remain the source of goals, ideas, and ethical principles that guide the development of models. The future is a partnership between humans and artificial intelligence, in which humans define objectives, and AI systems provide the means to achieve them, unlocking new horizons of imagination and possibility.

Thus, the goal of this article is to formalize no-code system creation through prompt engineering, analogous to fundamental programming language constructs. We introduce *"building blocks"* (primitives) such as conditional operators, loops, and functions—described in natural language and mathematically formalized. Additionally, we explore the composition of these primitives for constructing complex systems, such as semantic networks, and propose the concept of a *"prompt language"* as a domain-specific language (DSL) for no-code AI engineering.

The primary objectives of this research include:

- Developing mathematical models for conditional operators, loops, and abstractions that enable natural language-based system control.
- Introducing the concept of a "prompt language", defining principles for structuring and composing prompts, allowing for AI system management through textual instructions.
- Implementing adaptive optimization methods. Applying gradient descent and iterative refinement techniques to enhance prompt efficiency.
- Presenting the "swarm of virtual experts" model, examining how multiple LLMs can collaborate to overcome the limitations of individual models through result aggregation.

Although the topic of prompt engineering is relatively new, it has roots in many modern areas of scientific research:

The Concept of Cellular Automata (Stephen Wolfram):
In *A New Kind of Science* [Wolfram, 2002], Stephen Wolfram proposed the idea that simple rules can generate complex behavior. This concept has proven analogous to our approach to prompt engineering, where basic primitives (conditions, loops) form complex systems.

Programming by Example (PBE):
Research in PBE [Gulwani, 2016] demonstrated that humans can interact with systems by providing examples rather than detailed code. The work describes a methodology for program synthesis based on input-output examples, enabling non-programmers to automate repetitive tasks without writing traditional code.

Domain-Specific Languages (DSL):
Existing work on domain-specific languages

provided a theoretical foundation for developing a "prompt language" tailored to a no-code approach. Specifically, the study [Hermans, 2009] generates WCF/C# code for service-oriented software based on data contract definitions between clients and servers.

Gradient Descent for Prompt Optimization: Recent research [Reed et al., 2022] demonstrated the applicability of gradient descent for improving prompts, which became a key part of our iterative debugging approach.

Aggregation of Predictions from Multiple Models:
The concept of ensembles and voting [Lande, 2024] underpins our "swarm of virtual experts" model, where predictions from multiple LLMs are aggregated to produce more accurate conclusions.

2. Formal Definitions of Basic Primitives

Each primitive in the no-code prompt engineering framework is described as a mathematical object with clear rules for transforming inputs into outputs. These primitives serve as fundamental "building blocks" for constructing complex logical structures, analogous to programming language operators but grounded in natural language.

2.1. The "Condition" Primitive (If-Else)

Formalization:

Let:

- Input — input data (e.g., text, numeric parameter, etc.).
- C — a condition (predicate) that returns True or False.
- A_{True}, A_{False} — two possible actions executed when the condition evaluates to True or False, respectively.

The prompt function P is defined as:

$$P(Input) = \begin{cases} A_{True}, & if \quad C(Input) = True \\ A_{False}, & otherwise \end{cases}$$

This formal definition mirrors the classical if-else operator in programming languages but is implemented via textual instructions.

Explanation:

The "Condition" primitive works by instructing the LLM to execute one of two scenarios based on the evaluation of a logical condition. The condition can be any Boolean construct, such as:

- Checking for the presence of a specific word in the text.
- Comparing numeric values.
- Analyzing data context or categories.

Example:

"If the text contains the term 'cybersecurity,' return its definition; otherwise, provide a list of related terms."

Here:

- C — presence of the word "cybersecurity" in the text.
- A_{True} — generating the definition of "cybersecurity."
- A_{False} — retrieving associations for "cybersecurity."

This primitive enables the model to make decisions based on predefined criteria, effectively simulating logical branching.

2.2. The "Loop" Primitive (For-Loop)

Formalization:
Let:

- $S = \{s_1, s_2, ..., s_n\}$ — the set of elements to be processed.
- F — the operation applied to each element of the set.

The prompt function $P(S)$ is defined as the union of the results of applying the operation FFF to each element of the set:

$$P(S) = \bigcup_{i=1}^{n} F(s_i).$$

This formal definition is analogous to the classical for loop in programming languages, where the same operation is repeated for all elements of a set.

Explanation:

The mechanism of the "Loop" primitive works by passing a list of elements along with an operation to be applied to each element to the LLM. The result is the aggregation of all obtained responses. This approach is useful for large-scale data analysis, pattern extraction, or generating responses for multiple inputs simultaneously.

Example:

"For each term from the list [′phishing′,′firewall′], find 3 examples of its usage."

Here:

- S = {[′phishing′, ′firewall′} — the set of terms.
- F — the operation of finding examples of a term's usage.

The result will be the union of examples for both terms.

2.3. Primitive "Function" (Abstraction)

Formalization:

Let:

- F: $X \to Y$ be a function that transforms elements from set X into elements of set Y.
- x be an input element**.**
- *parameter* be a set of parameters that regulate the function's behavior.

The prompt-function $F_{extract}(x,parameter)$ is implemented through an instruction passed to the model:

$F_{extract}(x,parameter)$ =

= *Promt*(x, *instruction with parameter*).

This formal definition is analogous to abstraction in programming languages, where a function can accept parameters to adjust its logic.

Explanation:

The mechanism of the "Function" primitive works by passing an LLM a specific task with parameters that define the details of its execution. This enables the creation of flexible instructions that can be reused for different inputs.

Example:

"*Extract all terms from the text that belong to the category {category}."*

Here:

- *parameter*={*category*} is specifies which category of terms should be extracted.
- x is the text from which the terms should be extracted.

For example, if the category is "cybersecurity," the model will extract only technical terms related to this topic.

General Principles of Primitives

1. **Clarity of Formulation:** Each primitive must be precisely described to ensure unambiguous understanding by the model.
2. **Flexibility:** Primitives can be parameterized, allowing them to be adapted for different tasks.
3. **Composition:** Primitives can be combined to create complex systems, similar to how programmers write code using basic constructs.

These three primitives — "Condition," "Loop," and "Function" — form the foundation for no-code system creation through prompt engineering, allowing natural language to be used as a tool for managing AI system logic.

3. Composition of Primitives into Systems

A system is an orchestration of primitives, analogous to programs**.**

3.1. Prompt Syntax as a Language

Formal Grammar:

A prompt can be represented as an Abstract Syntax Tree (AST):

Prompt ::= *Primitive* | (*Prompt* $\oplus$ *Prompt*) | *Condition*(*Prompt, Prompt*)

where $\oplus$ denotes composition.

3.2. Example: Building a Semantic Network

Task: Automatically create a graph $G=(V,E)$, where V represents terms and E represents relationships.

Step 1. Defining Nodes (V)

Primitive: A loop for extracting terms.

$$V = P_{ExtractTerms}\left(Input\right) =$$
$$= \bigcup_{s \in S} F_{extract}\left(s, category = "cybersecurity"\right)$$

Step 2. Defining Edges (E)

Primitive: A condition for filtering relationships.

$E=\{(u,v) | u,v \in V, P_{checkRelation}(u,v)=\text{True}\}$,

where $P_{checkRelation}$ — is the prompt:

"Are the terms {u} and {v} related? Answer: Yes/No. "

Step 3. Classifying Edge Types

Primitive: A function with a parameter (relationship type).

type(e) = $F_{classify}$(e, *parameter* = {"attack","defense"})

4. Mathematical Model of the "Swarm of Virtual Experts"

4.1. Definition of the Swarm

A swarm is a set of N independent prompt functions $\{P_1, P_2, ..., P_N\}$ that solve a single task.

4.2. Aggregation of Results

Voting: For each result y_i:

$$y_{final} = \arg\max \sum_{i=1}^{N} \delta(y_i, y),$$

where δ is the Kronecker function.

Average value (for numerical data):

$$y_{final} = \frac{1}{N} \sum_{i=1}^{N} y_i,$$

4.3. Example: Defining Terms Through the Swarm

Prompt for 3 LLMs:

$$V_{final} = V_1 \cup V_2 \cup V_3 \setminus \{\text{terms that appear only once}\},$$

where V_i are the results of individual models

5. Iterative Tuning as an Optimization Problem

Iterative prompt tuning through gradient descent is a systematic approach to improving the quality of interaction with an LLM. It allows precise adjustment of the model's logic by modifying the instructions being passed. This approach is especially useful for no-code prompt engineering, where the primary goal is to use natural language as efficiently as possible to control complex systems.

5.1. Problem Formulation

The main goal of iterative tuning is to improve the quality of the result obtained from the LLM by changing the structure or phrasing of the prompt. Formally, let:

- Input — the input data for the model.
- P(Input) — a function that describes the prompt passed to the model.
- Output = P(Input) — the result produced by the model for the given prompt.
- Target — the desired (ideal) result that needs to be achieved.

The goal is to minimize the divergence between Output and Target, which can be measured using a quality metric d:

$$d(Output, Target) \to \min.$$

where d is the metric that defines the quality of the match between the model's output and the desired result. Examples of such metrics could be:

- F1-score: For classification tasks or term extraction.
- BLEU: For text translation tasks.
- Cosine Similarity: For comparing semantic vectors.
- MSE (Mean Squared Error): For numerical predictions.

5.2. Gradient Descent in the Prompt Space

Gradient descent is an optimization method used to find the minimum of a function. In our case, the function is the divergence metric d, and the parameters are the prompt itself, which we want to improve.

Step 1: Defining the "Derivative" of the Prompt

The first step is to understand how exactly we need to change the prompt in order to reduce the value of the metric d. This is analogous to the "derivative" in gradient descent, where we analyze how each part of the prompt affects the model's outcome.

The process can be divided into the following stages:

1. **Analyzing the Results**

First, we compare the output with the target to identify flaws or errors in the initial data.

2. **Identifying Problem Areas**

We determine which parts of the prompt might be causing incorrect model behavior. For example:

- Are the conditions clearly defined?
- Is there enough context provided?
- Are there unnecessary details that might confuse the model?

3. **Formulating Hypotheses**

Based on the analysis, we generate hypotheses about what changes in the prompt could improve the result.

Step 2: Correcting the Prompt

Once we have identified how to change the prompt, we apply these changes. The correction process can be formalized as:

$$P_{new} = P_{old} - \alpha \cdot \nabla d(P_{old}),$$

where:

- P_{new} is the new prompt after correction,
- P_{old} is the current prompt**,**
- α is the "learning rate," which determines how significantly we change the prompt in one step,
- $\nabla d(P_{old})$ is the direction in which the prompt should be adjusted to reduce the metric d.

In practice, this process looks like this:

- Adding details: If the model does not understand the context, a more specific explanation or examples can be added.
- Reducing excess: If the model is overloaded with information, the prompt can be simplified.
- Changing the format: Often, changing the format of the output data (e.g., requesting the result in JSON format) helps make the response more structured.

Example:

Let's consider an example of iterative debugging of a prompt for extracting technical terms from text.

1. Initial prompt: "Extract terms from the text." The result may be insufficiently accurate because the model might not understand which specific terms should be extracted.
2. Correction 1: Adding context "Extract technical terms from the text related to cybersecurity." Now the model has more information about the category of terms.
3. Correction 2: Specifying the standard "Extract technical terms from the text using the ISO/IEC 2382:2015 - Information technology standard." This version of the prompt defines the criteria for extracting terms even more clearly.
4. Correction 3: Forming the response format "Extract technical terms from the text using the ISO/IEC 2382:2015 - Information technology standard. Present the result in JSON format." This format requires the model to provide a more structured output.

Each debugging step reduces the value of the metric d. The process continues until the discrepancy between Output and Target becomes sufficiently small or until further improvements become negligible.

Iterative debugging can be represented as a sequence of prompts $P_1, P_2, ..., P_N$, where each subsequent prompt is an improved version of the previous one:

$$P_{k+1} = P_k - \alpha \cdot \nabla d(P_k).$$

Provided that the step α is chosen correctly and the metric d is differentiable, the process converges to the optimal prompt that minimizes the discrepancy between Output and Target.

6. Prompt Language for No-Code AI Engineering

A domain-specific language in the context of a no-code approach is not about syntactic constructions, but rather instruction templates in natural language that allow controlling system logic through structured dialogues with LLMs. Unlike classical DSLs, there are no keywords or grammar rules here — only principles of prompt composition that simulate programming.

6.1. Principles of "Prompt Language"

Modularity:
Each prompt performs an atomic task, similar to a function.

Example (prompt "Term Extraction"):

> Analyze the text and list all key technical terms. Output them in a JSON list with explanations.

Composition:
Prompts are combined sequentially, where the output of one becomes the input for another.

Example of a chain:

> [Text] → Prompt "Term Extraction" → [List of terms] → Prompt "Find Connections" → [Semantic Network]

Parameterization:
Variables in curly braces make prompts flexible.

Example:

> Find {number} examples of the term {term} used in the context of {topic}

Conditional Logic:
Decisions are made based on LLM responses.

Example:

> Step 1: "Does the text contain the term 'neural network'? Answer yes/no."
> Step 2: If "yes" → run prompt "Explain neural network," otherwise → skip.

Typical examples of prompts based on these principles are provided in Appendix A.

6.2. Visual Programming through Prompts

No-code platforms allow composing prompts in the form of flowcharts.

Example of Interface:

> Block "Upload Text"
> → Block "Highlight Terms" (parameter: category="cybersecurity")
> → Block "Find Connections" (parameter: depth=2)
> → Block "Export to CSV"

Advantages:

- Users drag and drop blocks, adjust parameters in the form.
- The system automatically generates prompts "under the hood."

6.3. Formalization as a Task Graph

The system can be represented as a directed graph $G = (V, E)$, where:

- $V = \{v_1, v_2, ..., v_n\}$ are nodes-prompts;
- $E = \{(v_i, v_j) |$ The output of v_i is the input of $v_j\}$ are connections.

Mathematical model of execution:

- For each v_i, the result $R_i = LLM(v_i, Input_i)$ is generated.
- The system is a composition: $R_{final} = R_n \oplus ... \oplus R_1(Input)$.

6.4. Example: Semantic Network via Prompt Chain

Task:
Automatically create a graph $G = (V, E)$, where:

- V is the set of terms.
- E is the set of connections between terms.

The graph should describe the key concepts from a text on "quantum computers" and their interrelationships..

Step 1: Define Nodes (V)

Primitive: "Cycle".

Goal — extract all key technical terms from the text.

Prompt:

> Select 10 key concepts from the text on quantum computers. Name them in order of importance.

Example output:

V={ "qubit", "quantum entanglement", "superposition", "quantum algorithm", "quantum slowdown", "quantum bit", "quantum processor", "quantum measurement", "quantum tunnel", "quantum simulator"}

Step 2: Define Edges (E)

Primitive: "Cycle" + "Condition".

Goal — find connections between each term in *V* and other terms.

Prompt:

> For each term in the list {terms}, find 2 connections with other concepts. Types of connections: part-whole, property, application. Format: JSON.

Example output:

```
{
  "qubit": [
    {"type": "property", "connection":
"superposition"},
    {"type": "application", "connection":
"quantum algorithm"}
  ],
  "quantum algorithm": [
    {"type": "property", "connection": "quantum
measurement"},
    {"type": "application", "connection":
```

```
"quantum simulator"}
  ],
  "superposition": [
    {"type": "part-whole", "connection":
"qubit"},
    {"type": "property", "connection": "quantum
slowdown"}
  ],
  ...
}
```

Step 3: Classification of connection types

Primitive: "Function".

Goal — classify the types of connections for further visualization.

Prompt:

```
Describe the connection type between "{u}"
and "{v}". Format: JSON.
```

Example output:

```
{
  "qubit-superposition": "property",
  "qubit-quantum algorithm": "application",
  "superposition-qubit": "part-whole",
  ...
}
```

Step 4: Network Visualization

Primitive: "Function".

Goal — create a visual presentation of the graph $G = (V, E)$.

Prompt for the visualization stage, asking for data in CSV format for easy upload into Gephi:

```
Present the result in two CSV files for import
into Gephi:

nodes.csv file for nodes:
Id column: Unique identifier of the term.
Label column: Term name.
edges.csv file for edges:
Source column: Identifier of the starting node.
Target column: Identifier of the ending node.
Type column: Type of connection (e.g., 'part-
whole', 'property', 'application').
Graph data:
{edges}
```

Example output:

nodes.csv:

```
Id, Label
qubit, qubit
superposition, superposition
quantum algorithm, quantum algorithm
quantum slowdown, quantum slowdown
quantum simulator, quantum simulator
```

edges.csv:

```
Source, Target, Type
qubit, superposition, property
qubit, quantum algorithm, application
superposition, quantum slowdown, part-whole
quantum algorithm, quantum simulator,
application"
```

This approach will provide a simple way to create a graph in Gephi using a no-code approach to prompt engineering (Fig. 1).

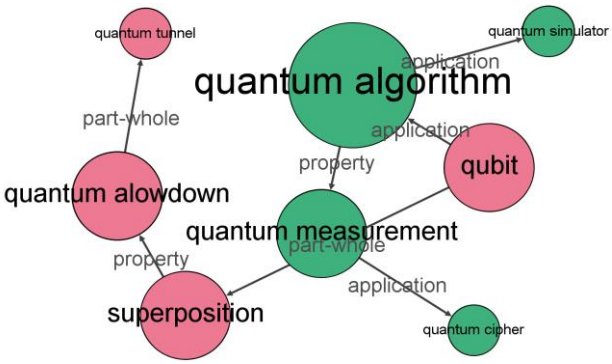

Figure 1: Result of loading the obtained data into the Gephi system

6.5. Advantages of "Prompt Language" over Traditional DSL

- Zero entry threshold, meaning the user doesn't need to learn syntax—just describe the task.
- Adaptability, which includes the ability to easily change logic by editing text instructions.
- Integration with AI tools, allowing prompts to be used in ChatGPT, Midjourney, or Claude without developing a separate interface.

6.6. Problem Limitations

1. Ambiguity of instructions

Solution: Adding examples to the prompts (few-shot learning), for example:

```
Create a list of terms, as in the example:
  – Text example: 'Qubit — the foundation of
    quantum computing.'
  – Response example: ['qubit', 'quantum
    computing'].
```

2. Vulnerability to changes in LLM

Solution: Providing container templates with parameters, for example:

Always use this structure:
{
'term': '...',
'connections': [{'type': '...', 'target': '...'}]
}

7. Examples of applications

We will show some examples of the proposed framework's application in the legal field, specifically in the areas of parliamentary control and legislation. We will consider the following tasks:

- "Automatic detection of risks, contradictions, or corruption threats in the text of bills";
- "Preparation of summaries for deputies regarding specific bills";
- "Analysis of the bill's text for possible lobbying interests".

7.1. Automatic detection of risks, contradictions, or corruption threats in the text of bills

A prompt is created that uses a loop for processing articles, conditions for risk classification, and functions for verifying legislation. The aggregation of results through a swarm of experts and iterative optimization ensures high accuracy, after which an output report is generated, structured for further analysis by parliamentary committees.

Input data:

- Text of the draft law *D*;
- Database of current legislation *L*.

Output data:

- Report *R*, containing:
  - A list of risks R_{risk}.
  - Conflicts $R_{conflict}$.
  - Corruption threats $R_{corrupt}$.
  - Recommendations $R_{recommend}$.

Primitives and their formalization:

Primitive "Loop" (For-Loop)

$$P_{loop}(D) = \bigcup_{s \in D} F_{analysis}(s),$$

where *s* is an article/section of the draft law, $F_{analysis}$ is the verification function.

Example:

"For each article of the draft law, perform an analysis for ambiguous terms."

Primitive "Condition" (If-Else)

$$P_{cond}(c) = \begin{cases} Add \;\; to \;\; R_{risks}, \;\; if \;\; C_{fuzziness}(s) = True, \\ Add \;\; to \;\; R_{key}, \;\; if \;\; C_{importance}(s) = True, \\ Other, \;\; otherwise. \end{cases}$$

Example:

"If the article contains the phrase 'may be established,' classify it as a corruption risk."

Primitive "Function" (Abstraction)

$$F_{expert}(s, parametr) =$$

$$= \begin{cases} \text{Term extraction}, \;\; if \;\; \text{parameter} = \text{"terminology"}, \\ \text{Compliance check}, \;\; if \;\; \text{parameter} = \text{"legislation"}. \end{cases}$$

Example:

"The 'Compliance Check' function compares the article with the Constitution and the 'Public Service Law.'"

System composition

Syntax*:*

$$System = P_{loop} \oplus P_{cond} \oplus F_{expert}.$$

Stages*:*

1. Breakdown into articles:

   $S = Split(D)$ (*Loop*).

2. Analysis of each article:

   - Extract terms:
     $T = F_{expert}(s, \text{"}terminology\text{"})$.
   - Check for ambiguity:
     $C_{ambiguity} = True, \;\; if \;\; \exists t \in T:$
     $t = \text{"rational term"}$.
   - Check for contradictions:
     $C_{contradiction}(s, L) = True,$
     $if \;\; s \;\; contradicts \; L.$

3. Aggregation of results (Swarm of experts):
   - Use three LLMs to analyze each article.
   - Voting on results:
     $$R_{final} = \arg\max_{r} \sum_{i=1}^{3} \delta(r_i, r).$$

4. Example of a specific prompt

Task:
"Analyze the draft law for corruption threats."

Prompt:

Step 1: Split the text into articles.
Step 2: For each article:
- Extract terms (e.g., 'discretionary powers').

- Verify compliance with the Constitution (articles 8, 28).
- If the phrase 'without justification' is found, add it to $R_{corrupt}$.

Step 3: Report:

- A table with columns: Article, Problem, Recommendation.

Output Report:

Article	Art. 5
Issue	Discretionary powers without criteria
Recommendation	Add clear decision-making criteria
Article	Art. 12
Issue	Contradiction with the "Budget Law"
Recommendation	Align terms with the Budget Code

5. Optimization via Gradient Descent

Problem Statement:

$$d(R_{final}, R_{target}) \to \min,$$

where R_{target} — reference report.

Iteration 1:

- Prompt: "Highlight terms" → result: 70% accuracy.
- Correction: "Highlight terms related to corruption (e.g., 'tender', 'discretion')" → 90% accuracy.

6. Visualization (Gephi)

Prompt for CSV generation:

Create nodes.csv (columns: Id, Label) and edges.csv (Source, Target, Type) for the nodes 'Risk', 'Contradiction', and the connections between them.

7.2. Preparation of summaries for deputies regarding specific bills

The corresponding prompt should use a loop to process articles, conditions to identify risks, and functions to structure the output. Aggregating results through a swarm of experts (e.g., 3 LLMs) ensures accuracy. The final summary should be adapted for legislators: concise, with a focus on key aspects and problematic areas.

A concise, clear, and comprehensible summary of the bill must be created for legislators, highlighting:

- The main objective.
- Key provisions.
- Potential risks/contradictions.
- Recommendations for revision.

Input Data:

- Bill text *D*.
- Database of existing legislation *L*.

Output Data:

- Summary R in table/text format with sections: Objective, Key Provisions, Risks, Recommendations.

Primitives and Their Formalization

Primitive: "Loop" (For-Loop)

$$P_{loop}(D) = \bigcup_{s \in D} F_{analysis}(s),$$

where *s* represents an article/section of the bill, and $F_{analysis}$ is the function for extracting key elements.

Example:

"For each article of the bill, extract the main requirement and penalties."

Primitive: "Condition" (If-Else)

$$P_{cond}(c) = \begin{cases} Add \ \ to \ \ R_{risks}, \ \ if \ \ C_{fuzziness}(s) = True, \\ Add \ \ to \ \ R_{key}, \ \ if \ \ C_{importance}(s) = True, \\ Other, \ \ otherwise. \end{cases}$$

Example:

"If an article contains the phrase 'may be established,' classify it as a risk."

Primitive: "Function" (Abstraction)

$$F_{summary}(D, parametr) =$$

$$= \begin{cases} \text{Concise description}, \ \ if \ \ \text{parameter} = \text{"goal"}, \\ \text{Table of key provisions}, \ \ if \ \ \text{parameter} = \text{"structure"}. \end{cases}$$

Example:

"Create a table with columns: 'Article,' 'Requirement,' 'Responsible Authority'."

System Composition

Syntax:

$$System = P_{loop} \oplus P_{cond} \oplus F_{summary}.$$

Stages:

1. Splitting into articles:

   $S = Split(D)$ (*Loop*).

2. Analysis of Each Section:

   - Identifying the Goal: $Goal = F_{summary}(D, "goal")$.
   - Key Provisions:

$$Key = \bigcup_{s \in S} P_{cond}(s).$$

– Risks:

$$R_{risks} = \{ s \mid C_{fuzziness}(s) \vee C_{contradiction}(s, L) \}.$$

3. Summary Formation:

Utilizing $F_{summary}$ for structured output generation.

4. Example of a Specific Prompt

Task:

"Prepare a summary of the bill 'On Environmental Supervision' for deputies."

Prompt:

Step 1: Identify the goal of the bill (first three articles).
Step 2: For each article:
- Determine the main requirement.
- If there are sanctions, add them to the "Key Provisions" table.
- If a term is not clearly defined (e.g., 'environmental safety'), add it to the "Risks" section.

Step 3: Check compliance with the "Environmental Protection Act."
Step 4: Create a summary in the following format:
- Goal: [brief description].
- Key Provisions: table.
- Risks: list.
- Recommendations: [proposals for improvement].

Final Output:

Section	Content
Goal	Ensuring environmental safety through enhanced supervision of enterprises.
Key Provisions	
Article 5	Mandatory certification for industrial facilities.
Article 8	Fines for violating environmental standards (from 100,000 UAH).
Risks	Undefined term "environmental safety" (Article 2).
Recommendations	Clarify certification criteria and align them with international standards.

5. Optimization via Gradient Descent

Problem Statement:

$$d(R_{summary}, R_{etalon}) \to \min,$$

where R_{etalon} is the expert summary sample.

Iteration 1:

Prompt: "Identify the goal" → Result: too general.

Correction: "Identify the goal, specifying concrete objectives (e.g., 'reduce emissions by 20%')."

6. Visualization (Gephi)

Prompt for Generating CSV:

Create a file nodes.csv (columns: Id, Label) for the nodes 'Goal', 'Risk', 'Key Provision'

7.3. Analysis of the bill's text for possible lobbying interests

In the generated prompt, the following should be used: a loop for processing articles, conditions for detecting lobbying indicators, and functions for checking legislation. Aggregation through a swarm of experts and iterative optimization should ensure high accuracy. The output report must contain specific recommendations for parliamentary committees.

Objective:

To identify the following in the text of a legislative proposal:

- Provisions that grant advantages to specific groups/entities.
- Ambiguous wording that could be used for lobbying.
- Conflicts of interest (e.g., lack of transparency requirements).

Input Data:

- Legislative proposal text *D*.
- Database of lobbying practices *L* (e.g., "laws with exemptions for specific industries").

Output Data:

- Report R containing:
  - A list of suspicious provisions $R_{lobbying}$.
  - Conflict analysis $R_{conflict}$.
  - Recommendations $R_{recommendations}$.

Primitives and Their Formalization

Primitive: "Loop" (For-Loop)

$$P_{loop}(D) = \bigcup_{s \in D} F_{analysis}(s),$$

where s is an article/section of a bill, and $F_{analysis}$ is a function for detecting lobbying indicators.

Example:

"For each article of the bill, check for the presence of phrases like 'special conditions' or 'exceptions for'."

Primitive "Condition" (If-Else)

$$P_{cond}(c) = \begin{cases} Add \quad to \quad R_{lobbying}, \quad if \quad C_{lobbying}(s) = True, \\ Add \quad to \quad R_{conflict}, \quad if \quad C_{conflict}(s) = True, \\ Other, \quad otherwise. \end{cases}$$

Example:

"If an article contains 'benefits for entities in sector X', classify it as a lobbying provision."

Primitive "Function" (Abstraction)

$$F_{expert}(s, parameter) =$$
$$= \begin{cases} Term\ analysis, \quad if \quad parameter = "terminology", \\ Conflict\ checking, \quad if \quad parameter = "legislation". \end{cases}$$

Example:

"Check whether Article 5 contradicts the Law on Lobbying Transparency."

3. System Composition

Syntax:

$$System = P_{loop} \oplus P_{cond} \oplus F_{expert}.$$

Stages:

1. Splitting into articles:

   $S = Split(D)$ ($Loop$).

2. Analyzing each article:

   - Extracting suspicious terms:
     $T = F_{expert}(s, "terminology")$.
   - Checking for lobbying indicators:
     $C_{lobbying}(T) = True, \quad if \quad \exists t \in T:$
     $t = "Exemption\ for\ group\ A"$.
   - Checking for conflicts:
     $C_{conflict}(s, L) = True,$
     $if \quad "s \quad contradicts\ L"$.

3. Aggregation of results (Swarm of Experts):

   - Using three LLMs to analyze each article.
   - Voting on the results:
     $$R_{final} = \arg\max_{r} \sum_{i=1}^{3} \delta(r_i, r).$$

4. Example of a Specific Prompt

Task:

"Analyze the bill 'On Energy Regulation' for lobbying interests."

Prompt:

Step 1: Split the text into articles.
Step 2: For each article:
- Highlight terms like 'benefits', 'exceptions', 'subsidies'.
- If references to specific companies/sectors exist, add to $R_{lobbying}$.
- Check compliance with the Law "On Preventing Corruption" (Article 14).

Step 3: Report:
- Table with columns: *Article, Suspicious Provision, Conflict, Recommendations*.

Output Report:

Article	Art. 7
Suspicious provision	"Benefits for coal companies"
Conflict	Contradicts Art. 14 of the Law "On Preventing Corruption"
Recommendations	Remove references to specific companies
Article	Art. 7
Suspicious provision	"Exemptions for entities from Donetsk"
Conflict	*None*
Recommendations	Clarify criteria for regional exemptions

5. Optimization **via** Gradient Descent

Problem Formulation:

$$d(R_{final}, R_{etalon}) \to \min,$$

where R_{etalon} is the reference report.

Prompt: *"Identify benefits"* → result: 60% accuracy**.**
Correction: *"Identify benefits related to specific industries (energy, extraction)"* → 85% accuracy**.**

6. Visualization (Gephi)

Prompt for Generating CSV:

Create nodes.csv (columns: *Id, Label*) for nodes 'Lobbying Provision', 'Conflict', and links between them.

8. Conclusions

The developed no-code prompt engineering framework provides a formalization for working with artificial intelligence systems based on the following aspects:

The core idea of this work is to represent prompts as mathematically formalized analogs of fundamental programming language constructs (conditions, loops, functions). This transforms arbitrary textual queries into structured logical primitives that can be systematically composed, optimized, and analyzed. For example, a condition (If-Else) takes the form of a mathematical function with input parameters and output actions, a loop (For-Loop) is modeled as a multiple-processing operator, and a function (Abstraction) is defined as a parameterized data transformation process.

This approach enables the creation of complex logical systems through textual instructions, aligning with the paradigm of "programming by example" (PBE) but with greater precision and control.

The proposed "prompt language" differs from traditional domain-specific languages (DSLs) in its adaptability and lack of rigid syntax. It possesses properties of modularity, parameterization, and compositionality. Within this language, each prompt performs an atomic task (e.g., "extract terms" or "build relationships"), variables enclosed in curly brackets allow for logic adaptation without modifying the core structure, and prompts can be combined into task chains or graphs, analogous to functions in programs.

This lowers the entry barrier to programming for users, as it does not require coding skills while still providing flexibility for developing complex systems.

The framework introduces innovative techniques that expand the capabilities of prompt engineering, specifically gradient descent in the instruction space and the "swarm of virtual experts" model.

The gradient descent method formally models prompts as an optimization problem, where the quality metric is minimized through iterative text corrections. This allows for the automatic improvement of system efficiency, similar to machine learning processes.

Aggregating the results of multiple LLMs (via voting or averaging) within the "swarm of virtual experts" concept reduces the risk of errors characteristic of individual models. This approach follows ensemble principles in machine learning but is adapted for no-code AI interaction.

The analogy with Stephen Wolfram's concept of complexity emerging from simple rules is central to the novelty of this framework. Just as cellular automata generate nonlinear dynamic systems from basic structures, the proposed framework enables the construction of complex AI systems through the combination of three primitives. This establishes a new theoretical framework for understanding how textual instructions can simulate programming without code.

Compared to existing solutions, the proposed framework has the following differences:

- The proposed "prompt language," unlike a classical DSL, does not require learning new keywords or grammar but instead utilizes natural language templates.
- The mathematical formalization, in contrast to traditional prompt engineering, ensures statistical consistency of results, separating them from the "black box" of LLMs.
- The composition of primitives through text, unlike classical coding, expresses system logic more clearly for non-programmers than SQL or Python.

The proposed framework shifts the paradigm of AI interaction from a "black box" to a structured, formalized process. Its novelty lies in the fusion of mathematical precision, the efficiency of a no-code approach, and scalability through the composition of primitives. It is not just a development tool but also a new discipline — "prompt logic" — that paves the way for AI management accessible to everyone.

Bibliographic References

[Wolfram, 2002] Wolfram, Stephen. "New kind of science." (2002). Wolfram Media. 1197 p. ISBN: 9781579550080, 1579550088

[Gulwani, 2016] Gulwani, S. (2016). Programming by examples-and its applications in data wrangling. In Dependable Software Systems Engineering (pp. 137-158). IOS Press.

[Hermans, 2009] Hermans, F., Pinzger, M., & Van Deursen, A. (2009). Domain-specific languages in practice: A user study on the success factors. In Model Driven Engineering Languages and Systems: 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings 12 (pp. 423-437). Springer Berlin Heidelberg.

[Reed et al., 2022] Reed S, Zolna K, Parisotto E, Colmenarejo SG, Novikov A, Barth-Maron G, Gimenez M, Sulsky Y, Kay J, Springenberg JT, Eccles T. A generalist agent. arXiv preprint arXiv:2205.06175. 2022 May 12.

[Lande, 2024] Lande, Dmitry; Strashnoy, Leonard. Implementation Of The Concept Of A "Swarm Of Virtual Experts" In The Formation Of Semantic Networks In The Field Of Cybersecurity Based On Large Language Models. Available at SSRN: https://ssrn.com/abstract=4978924, DOI: 10.2139/ssrn.4978924 (Oct 17, 2024). - 15 p.

Appendix A: Prompt Templates for Typical Tasks

№	Task type	Prompt template
1	**Classification**	"Determine the category of the text: {text}. Options: {list}."
2	**Data Generation**	"Create {number} examples for {concept}. Conditions: {conditions}. Format: JSON."
3	**Error Analysis**	"Find and fix errors in the code: {code}. Explain each change."
4	**Text Translation**	"Translate the text from {source language} to {target language}: {text}."
5	**Information Structuring**	"Break the text into key points. Provide a brief description for each point: {text}."
6	**Question Generation**	"Generate {number} questions about the text: {text}. The questions should vary in difficulty."
7	**Comparative Analysis**	"Compare two concepts: {concept1} and {concept2}. Identify similarities and differences."
8	**Term Compatibility**	"Are the terms {term1} and {term2} compatible? Justify your answer."
9	**Expert Consultation**	"If you are an expert in {field}, what would you recommen to solve the problem {problem}?"
10	**Hypothesis Generation**	"Generate {number} hypotheses about the causes of the phenomenon {phenomenon}. Support with data or examples."
11	**Pattern Recognition**	"Identify patterns in the dataset: {data}. Present the results table format."
12	**Text Evaluation**	"Evaluate the quality of the text {text} based on the criteria clarity, logic, completeness. Provide recommendations."