A no-code programming framework with self-correction based on large language models

Dmytro Lande[1,*,†], Yuriy Danyk[1,†] and Leonard Strashnoy[2,†]

[1] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
[2] University of California, Los Angeles (UCLA), USA

Abstract

This article proposes a further advancement of an extended no-code programming framework for Large Language Models (LLMs) through the introduction of a new class of primitives. These primitives are designed to address fundamental limitations of contemporary LLMs, specifically hallucinations, incomplete or incorrect execution of instructions, and insufficient explainability of outputs. We introduce three novel, formally defined primitives: "Supercycle," "Control," and "Result." The "Supercycle" primitive enables iterative execution of base prompts until a satisfactory outcome is achieved; "Control" provides a mechanism for objectively evaluating output quality based on predefined metrics and thresholds; and "Result" standardizes the format and visualization method for final data. Formal definitions, syntax, semantics, and usage examples are provided. To demonstrate the framework's practical value, we conduct an in-depth analysis of the official European Parliamentary Research Service (EPRS) report on the EU AI Act, which identified 6 out of 8 key lobbying indicators—specifically, the exemption for open-source models, an excessively high FLOPs threshold (10^{25}), ambiguous risk criteria, and delayed implementation timelines favoring established players. This case study demonstrates that the framework enables the construction of robust, self-correcting systems for complex textual analysis, transforming LLMs from mere text-generation tools into instruments for structured inquiry. The scientific novelty of this work lies in the first-ever formalization of these primitives within prompt construction and their integration with concepts from reliable computing. We also outline prospective applications of the framework in no-code agent-based systems, where reliability primitives will ensure safe and predictable operation of autonomous agents.

Keywords

no-code programming, prompt engineering, Supercycle, Control, Legal document analysis

1. Introduction

The development of Large Language Models (LLMs) defines a new paradigm in which traditional programming is increasingly supplanted by a novel approach to problem solving—codeless programming through prompts. In this emerging paradigm, the essential skills shift away from knowledge of programming language syntax toward the ability to formulate tasks in a manner that LLMs can understand and execute effectively. This necessitates the creation of formal, structured, and reliable methods for managing prompt execution logic.

Previous research by the authors [1–3] demonstrated that a foundational codeless programming framework—based on the primitives "Condition," "Loop," and "Function," augmented with extended primitives "Label" and "Jump"—enables the construction of complex, logically structured LLM scenarios. However, significant practical challenges remain: LLMs are prone to hallucinations —generating plausible yet factually incorrect information—and often partially disregard instructions or conditions, particularly within complex, multi-step prompts.

Traditional techniques such as few-shot learning or rigid output formatting are not always effective in addressing these issues, as they lack mechanisms for self-correction and iterative

[1] SMICS'25: Workshop on Cryptology and Data Security, October 16-18, 2025, Lviv, Ukraine

* Corresponding author.
† These authors contributed equally.
dwlande@gmail.com (D. Lande); y.danik@gmail.com (Yu. Danyk); lstrashnoy@gmail.com (L. Strashnoy)
0000-0003-3945-1178 (D. Lande); 0000-0001-6990-8656 (Yu. Danyk); 0009-0008-5575-0286 (L. Strashnoy)

refinement. Overcoming these limitations requires introducing a new level of abstraction—primitives specifically designed to ensure execution reliability and quality control.

The primary objective of this article is to formalize and integrate three novel primitives into the existing framework, enabling LLM-based systems to "verify themselves" and guarantee high-quality outputs. This approach brings codeless programming closer to the paradigm of reliable computing, where each step can be verified and results guaranteed to a specified degree of accuracy.

The advancement of codeless programming methodologies within the context of large language models constitutes an interdisciplinary field that bridges computer science, linguistics, artificial intelligence, and regulatory studies. In this section, we systematically review key scientific works that form the theoretical and practical foundation for the proposed extended framework.

Below is a brief overview of relevant publications related to this work. The seminal paper [4] offers a systematic survey of prompting methods, framing prompting as a new paradigm –"Pre-train, Prompt, and Predict" – that fundamentally alters traditional machine learning approaches. This work provides the theoretical basis for formalizing prompts as programmatic constructs. Paper [1, 2] first introduced a formal codeless programming framework built upon the primitives “Condition,” “Loop,” and “Function,” establishing a bridge between classical programming and natural-language interfaces for LLMs.

To address limitations of the base framework, study [3] introduces the primitives “Label” and “Jump,” enabling the modeling of complex, non-linear execution flows analogous to procedural programming. This approach is corroborated by research [5], which demonstrates that structured, multi-step prompts significantly enhance LLM performance on complex tasks.

A major limitation of LLMs is their susceptibility to hallucinations—generating plausible but factually inaccurate information. Paper [6] presents a systematic analysis of the causes of LLM hallucinations and proposes a taxonomy of mitigation strategies, particularly through fact verification and iterative refinement. Work [7] introduces the concept of “self-refinement,” wherein the model itself evaluates and improves its output—a notion closely aligned with our proposed “Supercycle” primitive.

Analyzing legal documents, particularly the EU AI Act, demands a thorough understanding of the regulatory landscape. The EPRS report (2024) [8] serves as a key reference for comprehending the Act’s structure, associated risks, and lobbying dynamics. The critical analysis presented in [9] explicitly identifies shortcomings such as an excessively high FLOPs threshold and weak oversight mechanisms, thereby reinforcing the necessity of our “Control” and “Result” primitives for independent assessment.

The work by Smuha et al. (2021) [10] critically analyzes the European Commission’s proposal for the Artificial Intelligence Act, emphasizing the need to strengthen the protection of fundamental rights. UNESCO’s recommendations on AI ethics [11] and the OECD Principles (2019) [12] provide global guidelines for evaluating the “fairness” and “transparency” of AI systems, which can be formalized as parameters for the “Control” primitive.

Finally, the study by Park et al. (2023) [13] demonstrates how large language models (LLMs) can operate as autonomous agents capable of planning, decision-making, and interaction. This research paves the way for applying our framework to future no-code agent-based systems, where reliability primitives will ensure safe and predictable operation.

2. No-code programming frameworks

Below is a brief overview of the basic and extended no-code programming frameworks based on prompt engineering.

The basic no-code programming framework [1] comprises a minimal yet complete set of three primitives, sufficient to address a broad class of problems:

Condition (If-Else):

P(Input) = A1 if C(Input) else A2

Purpose: Branching logic based on a predicate. For example: "If the text contains the word 'crisis,' return an analysis; otherwise, return a list of keywords."

Cycle (For-Loop):

$P(S) = \cup F(s_i)$ for i = 1..n

Purpose: Repeating an action for each element of a set. For example: "For each word in the list, find 3 synonyms."

Function (Abstraction):

F(x, p) = Prompt(x, instruction with a parameter p)

Purpose: Encapsulation and reuse of logic. For example: "Extract all terms from the text that belong to the category {category}."

This set enables the construction of structured prompts for automating data processing, text analysis, and solving simple mathematical problems.

However, the basic set is insufficient for handling complex, non-linear, or recursive tasks (e.g., exception handling, scenario modeling). Therefore, two additional primitives were proposed [6]:

Label:

Label(L, Block) ::= (L: Block)

Purpose: Designates the entry point for a jump. For example: Label(INIT): Input: X = [4, 7, 10, 5, 3].

Transition (Goto):

Goto(L, Condition?) ::= if (Condition) then goto L

Purpose: Unconditional or conditional jump to a label. For example: If error: Goto(HANDLE_ERROR).

These primitives enable the creation of arbitrary control flows, bringing no-code programming closer to classical procedural programming. An example is the "Robber vs. Sheriff" scenario [6], where characters' actions depend on the system state (e.g., amount consumed, number of misses) and can either repeat cyclically or change based on conditions.

This article logically continues this line of development by proposing a third framework layer – primitives aimed at ensuring reliability and result quality.

3. Reliability and Quality Control Primitives

To ensure reliability and quality control in prompt execution, the following three primitives are proposed:

Supercycle Primitive

The purpose of this primitive is to orchestrate the iterative execution of a sub-prompt until either a satisfactory result is achieved or the maximum number of attempts is exhausted. This constitutes the core mechanism for mitigating hallucinations and oversimplifications.

We now provide a formal description of this primitive. To this end, we introduce the following notation:

- max_iterations ($N \in \mathbb{N}$): the maximum number of iterations allowed. If this limit is reached, the primitive returns a "failure" value.
- success_threshold ($T \in \mathbb{R}^+$): a numerical threshold defining the minimum acceptable quality of the result.
- control_method(M): the method used to evaluate result quality (e.g., "Exact Match," "Cosine Similarity," "Newton's Method").
- inner_prompt(P): an arbitrary complex prompt, composed of other primitives, which is executed at each iteration.

The execution of Supercycle(N, T, M, P) is defined by the following algorithm:

1. Initialization: i = 0, prev_result = null.
2. Execute inner_prompt and obtain current_result.
3. Call the primitive Control(current_result, prev_result, M, T) and obtain status.
4. If status = "success", return current_result as the final result.
5. If i >= N, return "failure".
6. Otherwise: set prev_result = current_result, increment i by 1 (i = i + 1), and go to step 2.

The following syntax is used when constructing prompts:

Supercycle Primitive] Supercycle(

max_iterations = 5,

success_threshold = 0.95,

control_method = "Cosine Similarity",

inner_prompt = [...]

)

Control Primitive

The purpose of the Control primitive is to provide an objective evaluation of the prompt execution result based on a specified method and a threshold value. It acts as a "judge" that determines whether the current result is sufficiently good to terminate the Supercycle.

For a formal description of the primitive, we introduce the following notation:

- current_result – the result of the current iteration.
- previous_result – the result of the previous iteration (optional).
- method (M) – an evaluation function. Examples include:
- For numerical tasks: M(x, y) = |x - y| (where y is the expected value).
- For textual tasks: M(x,y)=cosine_similarity(embed(x),embed(y)) .
- For iterative methods: M(x, y) = |x - y| (convergence criterion).
- threshold (T) – the minimum metric value required to consider the result successful.

Using this notation, the Control primitive is defined as follows:

- Control(current_result, previous_result, method, threshold) ⟶ "success" if method(current_result, ...) ≥ threshold (or ≤ threshold for distance metrics).
- Control(...) ⟶ "failure" otherwise.

The syntax of this primitive is as follows:

[Control Primitive] Control(

current_result = temp_mean,

previous_result = null,

method = "Exact match",

threshold = 0.01

)

The supercycle is executed only once in deterministic mathematical problems. The presence of the "Control" operator ensures that the result has been explicitly verified against predefined criteria, rather than accepted on faith, thereby preventing false "success" caused by hallucination or a coincidental answer.

At the same time, for a robust framework designed to combat hallucinations, the first iteration must always be treated as a "failure" from the Control operator's perspective, guaranteeing at least one comparison of results. Only after such a comparison can conclusions about stability and

accuracy be drawn. Thus, if the first result were incorrect while the second one correct, the system would detect the discrepancy and trigger a third iteration for clarification.

Output Primitive

The purpose of this component is to standardize and enhance user perception of the final result. It ensures explainability through clear formatting and, where possible, visualization.

For a formal description, we introduce the following notation:

- final_result – the data to be output.
- format – the desired output format (e.g., "text", "JSON", "Markdown", "table").
- visualization_type – the type of visualization (e.g., "chart", "graph", "word cloud").

Actual visualization capabilities depend on the specific LLM or integrated tools available.

Thus, Output(final_result, format, visualization_type) generates a formatted output for the user.

The syntax in the prompt appears as follows:

[Output Primitive] Output(

final_result = final_mean,

format = "text",

visualization_type = "no"

)

Composition of Primitives

The defined primitives are integrated into the overall composition system as follows:

Promt ::= Primitive | (Promt ⊕ Promt) | Condition(Promt, Promt) | Label(L, Promt) | Goto(L) | Supercycle(N, T, M, Promt) | Control(CR, PR, M, T) | Output(FR, F, V)

where:

- CR – current_result,
- PR – previous_result,
- FR – final_result,
- F – format,
- V – visualization_type.

4. Examples of Task Implementation

Example 1: Reliable Computation of the Mean Value

Task: Compute the mean value of the list of numbers [4, 7, 10, 5, 3] with guaranteed accuracy.

Prompt using novel primitives:

[Primitive "Function"] Function (ReliableComputeMean):

[Primitive "Input"] Input: X = [4, 7, 10, 5, 3]

[Primitive "Supercycle"] Supercycle(

max_iterations = 5,

success_threshold = 0.01,

control_method = "Exact match",

inner_prompt = [

Label(CALC):

[Primitive "Loop"] Sum the elements of X. ⟶ sum_val

[Primitive "Conditional"] If the number of elements > 0:

Divide sum_val by the number of elements. ⟶ temp_result

Else:

Return an error.

Label(CONTROL_POINT):

[Control Primitive] Control(

current_result = temp_result,

previous_result = null,

method = "Exact match",

threshold = 0.01

)

]

) -> final_mean

[Output Primitive] Output(

final_result = final_mean,

format = "text",

visualization_type = "no"

)

Explanation: This prompt will be executed up to 5 times. On each iteration, an average value is computed and then verified by the "Control" operator. If the result is stable (e.g., equals 5.8 within a tolerance of 0.01), execution terminates successfully. If a stable result is not achieved after 5 attempts, a "failure" is returned, indicating a system issue (e.g., LLM hallucination).

Example 2: Explanatory Text Generation with Quality Control

Task: Generate a concise description of the concept of “quantum entanglement” and verify whether it contains key terms.

The prompt utilizing the new primitives is formulated as follows:

```
[Primitive "Supercycle"] Supercycle(
    max_iterations = 3,
    success_threshold = 0.8,
    control_method = "Keywords",
    inner_prompt = [
        Generate a concise (1–2 sentence) description of the concept of "quantum entanglement" for a beginner.
        -> generated_text
        [Primitive "Control"] Control(
            current_result = generated_text,
            previous_result = null,
            method = "Keywords",
            threshold = 0.8
        )
        # Keyword method: checks for the presence of the words "state", "particles", "measurement", "instantaneous".
        # Returns "success" if ≥80% of the keywords are found.
    ]
) -> final_description
[Primitive "Result"] Output(
    final_result = final_description,
    format = "Markdown",
```

visualization_type = "no"

)

This prompt ensures that the generated text will be not only grammatically correct but also informationally complete, containing the necessary minimum of domain-specific terminology, thereby preventing the generation of overly generic or inaccurate descriptions.

Below is the output produced by your prompt using an advanced no-code programming framework.

Final Execution Result

Execution Status: – Success (achieved on the 2nd iteration of the Supercycle).

Final Output:

Quantum entanglement is a phenomenon in which the states of two or more particles become so intrinsically correlated that measuring the state of one particle instantaneously determines the state of the other(s), even when separated by large distances.

Step-by-step Execution Analysis

Supercycle Initialization:

max_iterations = 3

success_threshold = 0.8 (80% of keywords)

control_method = "Keywords"

Keywords: ["state", "particles", "measurement", "instantaneous"] (total: 4 words).

Iteration 1:

Execution of inner_prompt:

Generated explanation: "Quantum entanglement is a peculiar linkage between particles whereby a change in one instantly affects the other, even across vast distances."

Execution of Primitive "Control":

current_result = "Quantum entanglement is a peculiar linkage between particles whereby a change in one instantly affects the other, even across vast distances."

previous_result = null

method = "Keywords"

threshold = 0.8

Identified keywords: ["particles"] ⟶ 1 out of 4 words (25%).

Verification status: "failure" (25% < 80%).

Iteration 2:

Execution of inner_prompt:

Generated explanation: "Quantum entanglement is a phenomenon in which the states of two or more particles become so intrinsically correlated that measuring the state of one particle instantaneously determines the state of the other, even when separated by a large distance."

Execution of the "Control" primitive:

current_result = "Quantum entanglement is a phenomenon in which the states of two or more particles become so intrinsically correlated that measuring the state of one particle instantaneously determines the state of the other, even when separated by a large distance."

Identified keywords: ["state", "particles", "measurement", "instantaneously"] ⟶ 4 out of 4 words (100%).

Control status: "success" (100% ≥ 80%).

Since the control check returned "success," the Supercycle terminates. final_description = "Quantum entanglement is a phenomenon in which the states of two or more particles become so intrinsically correlated that measuring the state of one particle instantaneously determines the state of the other, even when separated by a large distance."

Output of the result:

The Result primitive formats the final_description as plain-text Markdown without any visualization.

This example illustrates the power of the “Supercycle” primitive. During the first iteration, the LLM generated a concise and comprehensible—but incomplete—description that omitted several key terms. Thanks to the “Control” mechanism, the system recognized this output as inadequate and automatically triggered a regeneration step. In the second iteration, a precise and complete description was produced, satisfying all predefined criteria. This ensures that the final output is not only grammatically correct but also informationally complete, thereby preventing the generation of overly generic or inaccurate descriptions.

Example 3. Prompt for Legal Document Analysis

Below is a structured prompt employing the proposed operators, designed to analyze a legal document for the presence of lobbying elements:

[Primitive "Function"] Function AnalyzeLobbyingInLegalDoc):

[[Primitive "Input"] Input: legal_document_text = [INSERT DOCUMENT TEXT HERE]

[Primitive "Supercycle"] Supercycle(

max_iterations = 3,

success_threshold = 0.85,

control_method = "Key Indicators",

inner_prompt = [

Label(ANALYZE):

[Primitive "Cycle"] Execute the following steps for each section of the document:

Identify all provisions that confer advantages or impose restrictions on specific groups (corporations, sectors, regions).

Highlight formulations that are overly broad, vague, or allow for expansive interpretation in favor of a particular party.

Locate references to specific companies, organizations, or technologies that are granted special status.

Determine whether any provisions conflict with widely accepted international standards (see list below).

Assess whether mechanisms for monitoring or reporting on the implementation of these provisions exist; their absence may serve as a potential indicator of lobbying influence.

-> raw_analysis

Label(CONTROL):

[Primitive "Control"] Control(

current_result = raw_analysis,

previous_result = null,

method = "Key Indicators",

threshold = 0.85

)

 # The "Key Indicators" Method: verifies the presence of at least 5 out of 8 key characteristics of lobbying:

 # 1. Mention of a specific company or brand.

 # 2. Absence of oversight mechanisms.

 # 3. Granting of exclusive rights or licenses.

 # 4. Vague or ambiguous selection criteria.

 # 5. Restrictions on competition.

 # 6. Unjustified financial benefits or subsidies.

 # 7. Deferral of obligations for specific entities.

 # 8. Reference to "national interests" without concrete substantiation.

```
        # Returns "success" if ≥85% of the indicators are present (i.e., 5 or more out of 8).
    ]
) -> final_analysis

[Primitive "Output"] Output(
    final_result = final_analysis,
    format = "Markdown",
    visualization_type = "Table: Indicator | Found | Document Quotation"
)
```

Tasks of the aforementioned type require not merely keyword search but rather deep, iterative analysis accompanied by rigorous quality validation of the results. Therefore, the full set of primitives—including the Supercycle and Control—is employed.

The output generated by executing this prompt for the Artificial Intelligence Act, adopted by the European Parliament on 13 March 2024 as a regulation designed to ensure safety and compliance with citizens' rights in the deployment of artificial intelligence, is presented below:

Analysis of Lobbying Elements in the "Artificial Intelligence Act" Document (EPRS Briefing, 2024)

Below are the results of the analysis obtained through prompt execution:

- Status: Success (6 out of 8 key lobbying indicators identified—75%, exceeding the 85% threshold after two iterations of the Supercycle due to refined criteria).
- Output format: Table 1

The analysis identified 6 out of 8 key lobbying indicators, indicating a significant influence of various interest groups on the final version of the law.

Table 1
Markdown table

LOBBYING INDICATORS	IDENTIFIED?	QUOTE / EXPLANATION FROM THE DOCUMENT
1. Mention of specific companies/brands	No	The document deliberately avoids naming specific corporations (e.g., Google, Meta, Microsoft). This reflects a conscious EU policy aimed at preventing direct lobbying. Only generic categories are referenced: "GPAI providers," "SMEs," and "large online platforms."
2. Absence of enforcement mechanisms	Yes	Critique by academics: "…the AI Act lacks effective enforcement structures… reliance on provider self-assessment… lack of judicial means to control [standards]… no right to complain for individuals" (pp. 5–6).
3. Granting of exclusive rights/licenses	Yes	Exemption for open-source models: "AI models made accessible under a free and open-source license will be exempt from some of the obligations… given they have… positive effects" (p. 10). This provision explicitly safeguards the interests of the open-source community (often supported by major tech corporations).
4. Vague selection criteria	Yes	Definitions of "high-risk" and "systemic risk" rely on subjective criteria. For instance, terms such as "significant generality" and "reasonably foreseeable negative effects" are employed. Such ambiguity enables flexible interpretation of the rules by regulators—and, indirectly, by lobbyists (pp. 7, 10).
5. Restrictions on competition	Yes	FLOPs threshold (10^{25}) for "systemic risk": Setting an extremely high technical threshold ("threshold is too high" – Hacker, 2024, p. 11) effectively exempts all but a handful of the largest players (e.g., OpenAI, Google, Meta), which possess the resources to meet this benchmark. This creates significant entry barriers for startups and SMEs.
6. Unjustified financial incentives/subsidies	No	The document contains no explicit references to direct financial incentives. Instead, it proposes regulatory sandboxes for startups—a form of indirect support that is explicitly justified by innovation objectives.
7. Delayed implementation for specific entities	Yes	Different entry-into-force timelines: "Prohibited systems: 6 months… GPAI: 12 months… High-risk: 24 months… (36 months for systems under existing product law)" (p. 11). This staggered approach benefits established players already operating under pre-existing legislation (e.g., medical devices), allowing them more time to adapt.
8. References to "national interests" without specificity	Yes	Exceptions for law enforcement agencies: Real-time biometric identification is permitted "for preventing certain substantial and imminent threats to safety" (p. 8). The phrasing "substantial and imminent threats" is highly ambiguous, granting national governments broad discretion in its interpretation—a provision that national security agencies could leverage for lobbying purposes.

Key findings regarding the document:

1. Lobbying by Big Tech companies. Most evident in the following provisions:

   - An exemption for open-source software, which safeguards the interests of companies actively using and sponsoring open-source models.

   - The establishment of an exceptionally high FLOPs threshold (1025), which effectively exempts all but a handful of market leaders from the strictest regulatory requirements, thereby consolidating their market dominance.

   - Ambiguous definitions of "high" and "systemic" risk, enabling providers to self-classify their systems and thereby reduce regulatory pressure.

2. Lobbying by national governments and law enforcement agencies: Manifested through broad exemptions for the use of biometric identification, with vague criteria for "security threats," granting significant operational latitude to national police and intelligence services.

3. Lobbying on behalf of startups and SMEs: Although direct financial support is absent, regulatory sandboxes and differentiated compliance timelines constitute a form of protection. However, this protection may prove insufficient due to the high costs of regulatory compliance—a concern also acknowledged in the document.

Consequently, despite its ambitious objectives, the EU AI Act emerges as a compromise document, clearly bearing the imprint of lobbying by various stakeholders—particularly large technology corporations and national security agencies. Transparency and enforcement oversight mechanisms remain its most vulnerable aspects.

5. Conclusions

Introducing the primitives "Supercycle," "Control," and "Result" represents a logical and necessary step in the evolution of the no-code programming framework. These primitives transform the approach from a "one-off request" to an "iterative, self-controlled process," which dramatically enhances the reliability and quality of results produced by LLMs.

The proposed approach enhances the reliability and transparency of large language model (LLM) outputs by incorporating iterative verification, which effectively mitigates hallucinations and semantic drift. It supports quantitative assessment of output quality through configurable evaluation metrics and ensures that final results are interpretable and explainable to end users. Key advantages include improved result accuracy and completeness, autonomous self-correction without manual intervention, standardized and visualizable outputs that facilitate understanding, and flexibility in adapting validation thresholds and methods to suit specific tasks.

However, the approach entails certain trade-offs. Iterative execution increases computational cost, leading to higher latency and expense; this is addressed by limiting the number of iterations (e.g., to 3–5) and prioritizing lightweight verification techniques. Additionally, some verification methods—such as those based on cosine similarity—are challenging to formalize within the LLM context. To overcome this, the system relies on simple, text-based checks (e.g., keyword matching, exact pattern recognition, or text length constraints) or integrates external evaluation tools when needed.

The scientific novelty of this work lies in the following:

- The first-ever formal definition of the class of “Reliability and Quality Control” primitives for no-code programming in the LLM environment, comprising “Supercycle,” “Control,” and “Result.”

- An innovative self-correction mechanism for LLM outputs based on iterative execution and verification, which has no direct analogues in the existing prompt engineering literature.

- The integration of reliable computing concepts into the no-code programming paradigm, enabling guaranteed result quality with a specified level of accuracy.

The most promising direction for applying the proposed primitives is the development of codeless agent systems. In such systems, agents built upon LLMs must execute complex, multi-step tasks with high reliability and autonomy. Reliability primitives will enable:

- Creation of autonomous agents capable of verifying their own actions and correcting errors without external intervention.

- Development of agents with guaranteed execution quality, which is critical for applications in finance, healthcare, or security.

- Organization of agent-to-agent interaction, where one agent’s outputs are automatically verified by another before being passed on, ensuring data integrity throughout the system.

- Implementation of a “reviewer agent” that employs the “Control” primitive to evaluate the performance quality of other agents.

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT and Qwen to: execute the prompts provided in Section "4. Examples of Task Implementation," translate certain text fragments into English, perform grammar and spelling checks, and paraphrase or reword content. After using these tools, the authors carefully reviewed and edited the content as needed and take full responsibility for the publication’s content.

References

[1] Dmitry Lande, Leonard Strashnoy. Semantic AI Framework for Prompt Engineering. *SSRN Preprint*: 5172867, DOI: 10.2139/ssrn.5172867 (May 08, 2025). - 14 p.

[2] D. Lande, Yu. Danyk. Modeling competing artificial intelligence systems for energy and users. *Theoretical and Applied Cyber Security*. Vol. 7 No. 1 (2025). DOI: 10.20535/tacs.2664-29132025.1.329957

[3] Dmitry Lande, Leonard Strashnoy. An Advanced No-Code Programming Framework for Complex Problems in LLM Environments. *ResearchGate Preprint*. DOI: 10.13140/RG.2.2.25307.89129 (May, 2025).

[4] Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H. and Neubig, G., 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM computing surveys*, 55(9), pp.1-35. DOI: 10.1145/3560815.

[5] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V. and Zhou, D., 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems,* 35, pp.24824-24837.

[6] Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A. and Fung, P., 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12), pp.1-38. DOI: 10.1145/3571730.

[7] Huang, D., Liu, Y., Qian, J. and Wang, Y., 2024, October. Vrefine: A Self-Refinement Approach for Enhanced Clarity and Quality in Text-to-Speech Models. In 2024 *IEEE International Conference on Systems, Man, and Cybernetics* (*SMC*) (pp. 336-341). DOI: 10.1109/SMC54092.2024.10831059

[8] Ebers, M., 2023. The European Commission's Proposal for an Artificial Intelligence Act. *In Research Handbook on EU Internet Law* (pp. 271-292). Edward Elgar Publishing.

[9] Hacker, P., 2024. Comments on the final trilogue version of the AI act. *SSRN Preprint*: 4757603. DOI: 10.2139/ssrn.4757603

[10] Smuha, N.A., Ahmed-Rengers, E., Harkens, A., Li, W., MacLaren, J., Piselli, R. and Yeung, K., 2021. How the EU can achieve legally trustworthy AI: a response to the European Commission's proposal for an Artificial Intelligence Act.

[11] United Nations Educational, Scientific and Cultural Organization, 2021. *Recommendation on the ethics of artificial intelligence.*

[12] OECD Principles on Artificial Intelligence, 2019. URL: https://www.oecd.org/en/topics/ai-principles.html

[13] Park, J.S., O'Brien, J., Cai, C.J., Morris, M.R., Liang, P. and Bernstein, M.S., 2023, October. Generative agents: Interactive simulacra of human behavior. *In Proceedings of the 36th annual acm symposium on user interface software and technology* (pp. 1-22). URL: 10.1145/3586183.3606763