Orchestration of No-Code Agents in the AgentFlow Framework

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)

Abstract

The AgentFlow framework introduces a new paradigm of no-code programming, where the system's logic is formulated in natural language through structured prompts, and execution is carried out by a large language model (LLM). However, due to the inherently sequential nature of LLMs, true parallelism, determinism, and reliable execution control are impossible without an external mechanism. This paper proposes the concept of a universal orchestrator – the "conductor" – as the single code component in the system, responsible for orchestrating, synchronizing, enabling true parallel execution, and integrating agents, while all domain-specific logic remains exclusively within the prompts. This approach preserves the philosophy of "One framework, countless scenarios," simultaneously ensuring accessibility for non-technical users and technical reliability. The limitations of existing solutions, such as LangChain and MetaGPT, are analyzed – these either require code writing or are constrained by an internal LLM-centric architecture. A formal model of the conductor, its architecture, and a Python implementation example are presented, confirming the practical feasibility of the approach. The proposed method establishes a foundation for building scalable, deterministic, and integrated no-code systems suitable for industrial applications.

Keywords: no-code programming, LLM, agent swarm, universal conductor, orchestration, AgentFlow, formal model, parallel execution, execution manager

Introduction

The advancement of large language models (LLMs) has given rise to new paradigms in human-computer interaction. Among these, no-code programming occupies a prominent place – an approach in which system functionality is defined not through syntactic constructs of programming languages, but through natural language. The AgentFlow framework [1], introduced in prior research, implements this concept by incorporating logical primitives expressed in natural language: Condition, Loop, Function, Label, and Goto, which structure agent behavior. Each

agent is defined as a tuple containing a role, state, logic, and communication rules, and a group of agents forms a swarm capable of simulating parallel task execution.

However, this simulation has fundamental limitations. By their architecture, LLMs are autoregressive mechanisms that generate responses sequentially. This means that even an explicit instruction such as Run_Parallel([...]) does not achieve true parallelism, but merely prompts the model to process different branches within a single thread. This approach leads to loss of control, non-determinism, limited scalability, and a lack of integration with real systems when actual process parallelization is required. These drawbacks become critical when transitioning from demonstrative examples to industrial applications.

To overcome these limitations, it proves beneficial to move beyond the LLM itself and introduce an external execution mechanism – a universal manager, a "conductor," responsible for orchestrating agents and ensuring true parallelism, synchronization, error handling, and system integration. This paper proposes the concept of such a universal manager – the only code-based component in the AgentFlow no-code agent programming system – responsible for executing all agents while keeping all domain-specific logic within the prompts. This approach preserves the no-code philosophy while simultaneously enabling reliable and scalable execution.

Related Work

The development of large language models (LLMs) has led to rapid growth in research on multi-agent systems, where each agent acts as an independent intelligent entity capable of decision-making, interaction, and execution of complex tasks. Within this context, several key platforms have been developed, each offering a distinct approach to the organization, orchestration, and execution of agent-based systems.

One of the most well-known and widely adopted platforms is LangChain [2]. It provides an extensive infrastructure for constructing chains of calls (chains), managing memory, tools (e.g., APIs, databases), and agents capable of making dynamic decisions. LangChain enables integration of LLMs with external systems, supports asynchronous execution, and includes modules for orchestration. However, this flexibility comes at the cost of a high entry barrier: each chain, agent, or tool must be implemented as code in Python or JavaScript. This makes LangChain a developer-oriented tool rather than one accessible to non-technical users, contradicting the no-code programming paradigm that AgentFlow aims to realize.

A similar approach is implemented in Microsoft Semantic Kernel [3], which enables "embedding" LLMs into traditional applications using plugins to perform tasks. Although it supports the concept of "skills" that resemble agents, these skills also require implementation in C# or Python. This further underscores the focus on programmers rather than domain experts.

In 2023, autonomous agent systems such as AutoGPT [4] and BabyAGI [5] emerged, using LLMs for planning, execution, and goal generation. These systems can decompose a goal into subtasks, execute them via tools, and monitor progress. However, they suffer from significant drawbacks: non-determinism, susceptibility to "goal drift," lack of clear structure, and high computational cost due to iterative LLM calls. Moreover, their logic is governed by the LLM's internal planning rather than a formal model, making them difficult to analyze theoretically or deploy reliably in mission-critical systems.

A more structured approach is proposed in MetaGPT [6], where multiple agents – each assigned a specific "role" (e.g., manager, developer, tester) – collaborate to solve tasks such as software generation. MetaGPT employs standardized roles and interaction templates, bringing it closer to the concept of an agent swarm. However, it still requires code for configuration, and its execution remains confined within the LLM without an external orchestration mechanism. This limits its scalability and integration capabilities.

Other studies, such as ChatDev [7], propose detailed interaction scenarios between agents (e.g., "designer speaks to developer"), which closely resemble the concept of agent communication in AgentFlow. However, these interactions are implemented through dialogues within a single LLM call, which again fails to achieve true parallelism or determinism.

Works [8–10] explore multi-agent system architectures based on task decomposition, communication, and self-improvement. Particular attention is given to memory mechanisms [10], which allow agents to retain experience, analyze errors, and improve performance over time. These ideas have been integrated into AgentFlow, where the agent's state is a formal component of the model. However, most existing systems implement memory through vector databases or local files, rather than a well-defined internal state explicitly passed between executions.

Thus, current platforms can be broadly divided into two groups:

1. Code-oriented (LangChain, Semantic Kernel) – highly functional, but requires technical expertise.

2. LLM-centric (AutoGPT, MetaGPT, ChatDev) – simplify interaction, but lack control, determinism, and scalability.

AgentFlow occupies an intermediate position: it preserves no-code accessibility for non-technical users by using natural language as the sole interface, while moving beyond LLM-centric architectures through the introduction of a universal "conductor" manager – an external orchestration mechanism that ensures true parallelism, determinism, and system integration. This enables a combination of the strengths of both worlds: formal, structured prompt logic and reliable, scalable execution.

It is essential to separately mention the no-code programming framework introduced in [11, 12], which is based on the idea that system logic can be entirely expressed in natural language through structured prompts, without resorting to traditional code. This allows domain experts – such as financial analysts, data scientists, and researchers – to build complex intelligent workflows using only textual instructions. The key to realizing this concept lies in the formalization of logical primitives that structure agent behavior and enable emulation of classical programming constructs.

The framework is built upon three core logical primitives: Condition, Loop, and Function. The Condition primitive enables branching based on a predicate, for example: "IF the sum of denominators equals zero, THEN return an error." The Loop primitive supports iterative operations over a set, for example: "*For each* x_i *in X, compute* $(x_i - \mu)^2$." The Function primitive allows abstraction of common operations, for example: "*Function*(*ComputeMean*): *calculate the average value.*" Together, these constructs form a minimal yet complete system for building analytical pipelines.

To support more complex scenarios – such as exception handling, time series analysis, or recursive algorithms – two additional primitives are introduced: Label and Goto. The Label primitive allows marking entry points in logical blocks, for example: "*Label*(*INIT*): *Input: X* = [4, 7, 10, 5, 3]." The Goto primitive enables jumps in the execution flow, for example: "*IF sum_denominator* == 0, *THEN Goto*(*HANDLE_ERROR*)." These primitives bring no-code programming closer to the procedural programming paradigm, providing precise control over the logical flow.

All primitives are mathematically formalized. For instance, Condition is defined as a function that takes a predicate C and two actions A_1 and A_2, returning A_1 if C is true, and A_2 otherwise. The composition of primitives enables the construction of complex logical structures: prompts may include nested conditions, loops within

functions, jumps between labels, and so on. This structured approach allows not only algorithm description but also ensures their deterministic interpretation by the LLM – a critical requirement for reliable execution.

The prompt creation process follows a well-defined methodology: (1) verbal formulation of the task, (2) clarification and addition of examples (few-shot learning), (3) structuring using logical primitives, (4) testing on benchmark cases, and (5) final refinement. This sequence transforms an unstructured query into a formalized, executable instruction that can be interpreted both by an LLM and by an external orchestration mechanism. This approach ensures high accuracy, adaptability, and seamless integration into industrial systems.

The Base AgentFlow Frameworks and Its Limitations

AgentFlow is based on the idea that a large language model can act as an interpreter of a programming system, where structured pseudocode in natural language replaces traditional syntax. Agents, organized into a swarm, execute individual parts of a task, exchange data, and collectively produce the final result. Agent states are preserved between invocations, enabling self-improvement mechanisms, while the formal model ensures determinism and enables theoretical analysis.

Despite these advantages, the base AgentFlow framework has significant limitations. Since execution occurs entirely within the LLM, it is inherently sequential. Even if a prompt contains an instruction such as "Run in parallel," the LLM generates the response in a linear order, leading to dependence on the context window, inability to synchronize processes, and absence of mechanisms for timeouts, queues, or error handling. Furthermore, as the number of agents increases, so does prompt length, quickly exceeding the model's context window. This renders the system unsuitable for scalable applications requiring integration with databases, APIs, or file systems.

One of the most widely used tools for building agent-based systems is LangChain – a platform that enables the orchestration of LLMs, tools, memory, and chains of calls. It supports asynchronous execution, includes built-in memory mechanisms, and allows integration with external systems. However, this functionality comes at the cost of requiring code. Each chain, agent, or tool must be implemented as a Python module, which contradicts the very principle of no-code development. For a non-technical user, configuring LangChain is an infeasible task, and the framework's complexity hinders maintainability and comprehension. LangChain is

a solution designed for developers, not for domain experts who wish to operate exclusively through natural language.

Anthropic subagents are specialized, purpose-built AI agents (sometimes called "personalities") that operate within Claude (or Anthropic's agentic systems) to handle specific tasks or domains [13]. Each subagent has its own configuration, including custom prompts, tool permissions, and an isolated context window, allowing it to function as a distinct "expert" for a particular workflow or problem area.

Key characteristics: Each subagent is focused on a clear, narrowly defined job (like code review, testing, research, or data wrangling), bringing domain specialization. Subagents operate in parallel, managed and coordinated by the main Claude agent, which delegates tasks and synthesizes results. They preserve their own conversation context, preventing cross-task confusion or context pollution, ensuring high relevance and accuracy in their domain. You control which tools, APIs, or system prompts each subagent uses, enabling security and precision for workflow automation.

In practice, Subagents are orchestrated by the lead agent, working together on projects—much like a team with distinct roles, improving both efficiency and output quality. They can automate complex or multi-step workflows by dividing tasks among specialized "teammates," each running independently and reporting results back to the orchestrator.

Subagents operate in parallel. They are managed and coordinated by the main Claude agent, which is responsible for delegating tasks to the subagents and synthesizing the results they produce. This allows for complex or multi-step workflows to be automated by dividing tasks among these specialized "teammates."

The Universal Manager "Conductor" as a Solution

To preserve the advantages of AgentFlow – accessibility, formality, and flexibility – while overcoming its limitations, we propose introducing a universal manager, the "conductor": a single program within the system responsible for executing all agents. This approach is based on a fundamental principle: all system logic resides in the prompts, with the sole exception being the code that handles execution.

The conductor contains no domain-specific logic. It does not know whether text is being analyzed or a correlation is being computed. Its role is solely to interpret a structured prompt-swarm, schedule execution, invoke agents via the LLM API, collect results, and return the final output. It is implemented as a static, unchanging

program (e.g., conductor.py), which remains constant regardless of the task being solved. This makes the conductor analogous to an interpreter or an operating system – a universal execution mechanism independent of any specific application.

The conductor enables true parallel execution, state management, synchronization, error handling, and integration with external systems – capabilities unattainable within a purely LLM-centric approach. The conductor does not "think" – it executes. All "thinking" remains within the prompts.

To rigorously describe this architecture, we introduce a formal model of the universal manager.

Let D be the universal "conductor" manager, defined as a four-component system:

$$D = \langle P, S, E, M \rangle,$$

where P is the prompt parser, S is the execution scheduler, E is the agent executor, and M is the state manager.

Each of these components is a function that operates on structured data but contains no domain-specific logic. They manipulate only the syntax and structure of prompts, not their semantic content. This ensures the system's universality: D can execute any swarm of agents, regardless of the application domain – be it text analysis, correlation computation, forecasting, or fact-checking.

The architecture of the universal "conductor" manager implements a strict separation between the execution mechanism and the system's domain-specific logic. This approach enables a single, unchanging execution infrastructure that remains constant across all tasks, while all functionality is defined exclusively through prompts. This establishes a paradigm analogous to a virtual machine model: just as the Java Virtual Machine (JVM) or a Python interpreter can execute diverse programs without modification, the "conductor" manager acts as a universal interpreter for agent swarms, ensuring stability, universality, and scalability [14]. Such an architecture forms the foundation for building deterministic, reliable, and user-friendly no-code systems accessible to non-technical users.

Let L be a formal language defining the syntax of prompts in AgentFlow. Each prompt $p \in L$ has the structure:

$$p = \langle type, id, role, logic, state, comm \rangle,$$

where $type \in \{AGENT, SWARM\}$ is the entity type, id is the unique identifier, role is the textual role of the agent, logic is the agent's behavior defined using

primitives: *Condition*, *Loop*, *Function*, *Goto*, state is the state dictionary $S = \{(k_i, v_i) \mid k_i \in K, v_i \in V\}$, and comm is the communication rules.

For an agent swarm, the following structure is introduced:

$$W = \langle name, A, L, D \rangle,$$

where $A = \{a_1, a_2, ..., a_n\}$ is a set of agents, L is the execution logic graph (control flow graph), and D is the input data (context).

The parser P transforms the input prompt p into an internal representation – an abstract syntax tree (AST):

$$P(p) = T_p,$$

where T_p is a tree whose nodes correspond to agents, execution branches, loops, etc.

For example, for the construct:

$$Run_{Parallel}([a_1, a_2]),$$

the AST takes the form:

$$T_{parallel} = Node(PARALLEL, children = [T_{a1}, T_{a2}]).$$

This corresponds to the control flow graph (CFG) model known in compiler technology [15], where each node represents an operation and edges represent control flow.

The scheduler S transforms the AST into a directed acyclic execution graph $G = (V, E)$, where V is a set of nodes (tasks) and E is a set of edges (dependencies). Each node $v_i \in V$ corresponds to an agent a_i, and an edge $(v_i, v_j) \in E$ indicates that a_j cannot begin execution until a_i has completed, i.e., $S(T_p) = G$.

For example, for the logical expression "*IF condition THEN: CALL* a_2", the graph contains a conditional edge:

$(v_{condition}, v_{a2}) \in E$ if and only if eval(*condition*) = *True*.

This corresponds to the workflow orchestration model used in systems such as Apache Airflow or Kubernetes [16], with the key difference that the execution graph is constructed not from code, but from natural language.

For asynchronous execution of agents, the executor E implements the function:

$$E(a_i, s_i, d_i) \rightarrow r_i,$$

where a_i is the agent, s_i is its state, d_i is the input data, and r_i is the result in JSON format.

The implementation of E involves constructing a prompt:

$$prompt_i = role(a_i)+logic(a_i)+state(s_i)+input(d_i),$$

and invoking the LLM via an API:

$$r_i = \text{LLM}(prompt_i).$$

For parallel execution, an asynchronous model is used:

$$E_{async}(V_{ready}) = \{E(v_i) \mid v_i \in V_{ready}\},$$

where V_{ready} is the set of nodes whose predecessors have completed.

This ensures true parallelism, in contrast to LLM-based emulation, and aligns with the asynchronous execution model in multithreaded systems.

The state manager M stores and updates agent states between executions. For each agent a_i, the state s_i is updated according to the rule:

$$s_i(t+1) = M(s_i(t), r_i),$$

where $s_i(t)$ is the state before execution, r_i is the result, and $s_i(t + 1)$ is the updated state.

For example, if an agent incorrectly performs a task, the system may add an entry to the state such as:

si.errors.append(task_id,ri.error)

or update the statistics:

si.accuracy=si.totalsi.correct.

The proposed model has analogies in other domains:

- The universal manager D resembles a virtual machine such as the Java Virtual Machine (JVM) or a Python interpreter, where D acts as the interpreter and the prompt serves as the program. However, the key difference lies in the input language: instead of a formal programming syntax, the system uses natural language as the medium for defining executable logic.
- Similarly, the execution graph G and scheduler S are analogous to components of workflow orchestration systems like Apache Airflow, where G corresponds to a directed acyclic graph (DAG) and S functions as the scheduler. The crucial distinction is that in the proposed model, the execution graph is derived from natural language descriptions rather than being explicitly coded. This enables non-technical users to define complex workflows using plain text, while preserving the rigor and determinism of structured execution.

Architecture and Implementation

Although the AgentFlow framework successfully emulates parallel agent execution through structured prompts, this emulation is inherently limited by the sequential nature of the large language model (LLM) architecture.

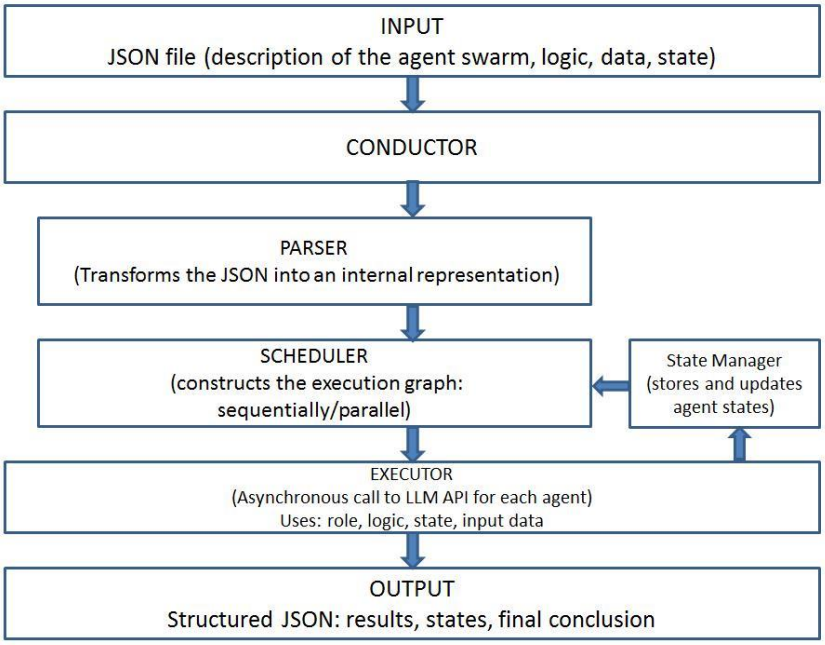

Figure 1: A multi-module system comprising four core components

To achieve true parallelism, determinism, synchronization, and integration with external systems, an external execution mechanism is required – a universal "conductor" manager. This component is the only code-based program in the AgentFlow system that remains fixed and unchanged, while all domain-specific logic is expressed exclusively through prompts. This section provides a detailed description of the conductor's architecture, including its formal model, block diagram, execution algorithm, Python implementation, and an example of executing a complex scenario that combines sequential and parallel execution.

The universal "conductor" manager is implemented as a multi-module system comprising four core components (figure 1).

The presented diagram clearly illustrates the data flow (from top to bottom) and the state feedback loop (via the State Manager), making the architecture transparent, understandable for a scientific audience, and suitable for inclusion in a research paper. The following notations are used in the diagram:

- **Input (top section):** The dispatcher receives a JSON file as input, containing the complete description of the agent swarm: the agents' roles,

logic, states, execution graph, and input data. This JSON serves as the single source of variability in the system.

- **Parser:** Transforms the JSON into an internal representation – an abstract syntax tree (AST) – enabling the system to understand the structure and dependencies of the task.
- **Scheduler:** Based on the AST, constructs the execution graph by determining which agents run sequentially, which in parallel, and which run under conditional branches. This graph is then passed to the Executor for processing.
- **Executor:** Asynchronously executes agents via the LLM API. For each agent, it constructs a prompt that includes the agent's role, logic, current state, and input data, then invokes the LLM to generate a response.
- **State Manager:** Works closely with the Executor. It provides the agent's current state before execution and updates it afterward based on the execution result. This mechanism enables self-improvement and memory persistence across iterations.
- **Output (bottom section):** The final output is a structured JSON object containing:
  - the output data from each agent,
  - the updated states of all agents,
  - and, if defined by the scenario, an aggregated conclusion generated by a moderator agent.

The execution algorithm of the dispatcher can be described as follows:

1. Load the prompt file (P)
2. Parse $P \rightarrow$ obtain the Swarm
3. Initialize the State Manager with S_0
4. Construct the execution graph based on L
5. For each step in the graph:

   5.1. Determine which agents are ready for execution
   5.2. If strategy = "Parallel" → launch asynchronously
   5.3. If strategy = "Sequential" → launch one after another
   5.4. For each agent:

   – Construct the prompt (role + logic + state + input data)
   – Call the LLM (via API)
   – Receive the JSON result
   – Update the agent's state
   – Pass the result to subsequent agents

6. When all agents have completed execution → return the final result

For the practical implementation of the proposed formal model of the universal "conductor" manager, a program was developed in Python, serving as the single code-based component within the AgentFlow ecosystem. This program implements the architecture described in the previous sections and ensures true agent orchestration through asynchronous execution, deterministic scheduling, and controlled state exchange. The program is designed according to the principle of "One framework, countless scenarios," making it universal, immutable, and independent of any specific application domain.

The program is implemented as a command-line script, conductor.py, which takes as input a JSON file containing the description of an agent swarm and returns a structured result in JSON format. The input file defines the agents, their roles, logic, state, and the execution graph composed of sequential, parallel, and conditional branches. Thus, all domain-specific logic remains outside the code, and the manager acts as a universal interpreter of these prompts.

The program's core architecture is built around four key components, corresponding to the formal model $D = \langle P, S, E, M \rangle$:

- Parser implements the function P by loading a JSON file and transforming its contents into an internal representation – objects of the classes Agent and Swarm. This process includes structure validation, state initialization, and construction of an abstract syntax tree (AST), which serves as the foundation for subsequent analysis.
- Scheduler implements the function S by analyzing the logic field in the input file, which contains a list of execution steps (e.g., Sequential, Parallel). Based on this list, an execution graph is constructed, where each step defines the strategy for launching agents. The system supports combinations of phases, enabling complex execution scenarios such as "sequential → parallel → sequential".
- Executor implements the function E through asynchronous calls to an LLM via API (in a real system – OpenAI, Anthropic, etc.). For each agent, a prompt is constructed, incorporating its role, logic, current state, and input data. Execution is implemented using the asyncio library, ensuring true parallelism when handling multiple agents. In the current implementation, LLM calls are simulated for demonstration purposes; however, the architecture allows seamless integration with any external API.
- State Manager implements the function M by preserving each agent's state between invocations. After receiving an execution result, the agent updates

its state – for example, by incrementing a call counter or recording error information. This enables self-improvement and adaptation mechanisms, which are essential for the long-term deployment of agents.

The program implements a deterministic execution algorithm: it processes each step of the graph sequentially, waits for the completion of all agents in the current phase (including those running in parallel), updates their states, and proceeds to the next step. The final output is a structured JSON object containing the results from all agents, their updated states, and execution metadata.

This architecture ensures high flexibility, scalability, and reliability, while keeping all domain-specific logic within the prompts. The program requires no modifications when adding new agents or changing tasks, confirming its role as a universal manager within the no-code programming paradigm.

Example of executing a complex scenario

To demonstrate the capabilities of the universal "conductor" manager, this section presents a complex scenario combining sequential, parallel, and dependent execution of agents. This scenario models a realistic analytical process commonly encountered in business analytics, financial modeling, or media monitoring, where data of different types must be integrated, processed independently, and then synthesized to produce a unified conclusion.

The goal of the scenario is to analyze the relationship between economic indicators and public sentiment as reflected in textual sources. The task involves identifying correlations between the dynamics of time series data (exchange rate, cryptocurrency price) and the emotional tone of news articles related to economic policy. Such analysis can help determine whether media sentiment reflects actual economic trends or whether there is a disconnect between official communications and their public perception.

The scenario is implemented using a swarm of four agents, organized into three logical execution phases:

Sequential Phase 1: Data Preparation.

Agent agent_1 ("Data Preparation") is executed first. Its task is to retrieve input data, verify its integrity, and structure it for further processing. This ensures quality control at the beginning of the process and guarantees that all subsequent agents operate on valid input data.

Parallel Phase: Independent Analysis of Different Data.

After successful data preparation, two agents are launched in parallel:

- agent_2 ("Time Series Analysis") – computes trends and seasonality in the time series data of the currency exchange rate and cryptocurrency price.
- agent_3 ("Text Analysis") – determines the emotional tone of the news article about the tax law, identifying whether it is presented neutrally, positively, or negatively.

This phase demonstrates true parallel execution, enabled by the dispatcher's asynchronous mechanism, ensuring efficient use of time and computational resources.

Sequential Phase 2: Integration and Conclusion Generation.

Upon completion of both parallel tasks, agent_4 ("Report Generation") is triggered. It receives results from all previous agents, compares the dynamics of economic indicators with the emotional tone of the news article, and produces a structured conclusion. This illustrates the dependency mechanism and result aggregation, which are essential for complex analytical systems.

This structure – *sequential* → *parallel* → *sequential* – is typical for many real-world applications, including ETL processes, analytical pipelines, and decision-support systems. It clearly highlights the advantages of the universal "conductor" manager: the ability to orchestrate complex execution graphs, ensure synchronization between phases, and integrate heterogeneous tasks into a unified workflow, while keeping all domain logic within the prompts.

The JSON file provided below contains the complete description of this scenario in a format interpretable by the "conductor" manager.

```
{
	"name": "complex_scenario",
	"agents": {
		"agent_1": {
			"id": "agent_1",
			"role": "Data Preparation",
			"logic": "Clean the data and convert it into a structured format.",
			"state": {}
		},
		"agent_2": {
			"id": "agent_2",
			"role": "Time Series Analysis",
			"logic": "Calculate trend and seasonality.",
			"state": {}
		},
		"agent_3": {
			"id": "agent_3",
			"role": "Text Analysis",
			"logic": "Determine the emotional tone of the news article.",
```

```
                        "state": {}
                },
                "agent_4": {
                        "id": "agent_4",
                        "role": "Report Generation",
                        "logic": "Combine all results into a final conclusion.",
                        "state": {}
                }
        },
        "logic": [
                {
                        "type": "Sequential",
                        "agents": ["agent_1"]
                },
                {
                        "type": "Parallel",
                        "agents": ["agent_2", "agent_3"]
                },
                {
                        "type": "Sequential",
                        "agents": ["agent_4"]
                }
        ],
        "data": {
                "time_series": [100, 105, 110, 115, 120],
                "news_text": "The company announced record profits."
        }
}
```


After launching the dispatcher: "python conductor.py complex_swarm.json", the following result is obtained:


```
{
        "swarm": " complex_scenario",
        "final_result": {
                "agent_1": {
                        "agent_id": "agent_1",
                        "result": {
                                "output": "executed",
                                "status": "success"
                        },
                        "state": {
                                "invocation_count": 1
                        }
                },
                "agent_2": {
                        "agent_id": "agent_2",
                        "result": {
                                "output": "executed",
                                "status": "success"
                        },
                        "state": {
```

```
                    "invocation_count": 1
                }
            },
            "agent_3": {
                "agent_id": "agent_3",
                "result": {
                    "output": "executed",
                    "status": "success"
                },
                "state": {
                    "invocation_count": 1
                }
            },
            "agent_4": {
                "agent_id": "agent_4",
                "result": {
                    "output": "executed",
                    "status": "success"
                },
                "state": {
                    "invocation_count": 1
                }
            }
        },
        "timestamp": "now"
}
```


This example demonstrates how the "conductor" manager can orchestrate complex scenarios by combining sequential and parallel execution, ensuring determinism and reliability.

Scientific and Practical Novelty

The scientific novelty of the proposed approach lies in the introduction of a universal execution mechanism for no-code agent systems, which clearly separates logic from execution. This enables formalization of the orchestration process, its mathematical definition, and ensures determinism – features unattainable within purely LLM-centric approaches. For the first time, we introduce the concept of a single code-based component ("conductor") that remains unchanged across all tasks, while all domain-specific logic is expressed exclusively in natural language. This establishes a foundation for theoretical investigation of no-code systems, including their stability, scalability, and self-organization.

The practical novelty consists in enabling non-technical users – experts in finance, medicine, education to build complex intelligent systems without writing code, while the underlying technical infrastructure remains stable, scalable, and integrable. The conductor can be embedded into a web interface, CLI, or API, serving as the core of a platform where users upload prompt files and the system

executes them with high reliability. This paves the way for widespread adoption of agent-based systems in real-world business processes.

Conclusions

The proposed AgentFlow framework establishes a new paradigm for interacting with large language models (LLMs), in which structured pseudocode in natural language – interpreted by the LLM as an execution system – replaces traditional programming. This approach enables non-technical users to formalize complex analytical processes without writing code, while simultaneously ensuring a high degree of control, structure, and output determinism. The introduction of logical primitives – Condition, Loop, Function, Label, and Goto – lays the foundation for building algorithmically sophisticated tasks, moving beyond simple question-answer interactions and positioning LLMs closer to the role of program logic interpreters.

The formal agent model, defined as a tuple comprising role, state, logic, and communication rules, allows for systematic organization of individual entity behavior within a swarm. This enables modular design of intelligent systems, where each agent acts as an independent processing module performing a clearly defined task. The concept of an agent swarm interacting according to a predefined execution graph provides an emulation of parallel execution – even within the inherently sequential architecture of LLMs. This emulation is based on explicit description of execution branches, allowing the LLM to process individual agents as independent entities and subsequently combine their results into a unified output. This approach ensures analytical coherence unattainable with unstructured prompts.

Particularly significant is the introduction of an agent internal state mechanism, serving as a form of memory that stores interaction history, accuracy statistics, and context from previous tasks. This enables self-improvement capabilities, where agents adapt their behavior based on past experience – an important step toward developing autonomous intelligent systems. The agent state is not a passive data store but is actively used in execution logic, enabling behavioral evolution over time. This creates the foundation for systems capable of real-time learning without external intervention.

To overcome inherent LLM limitations – such as the lack of true parallelism, execution control, and integration with external systems – we propose the concept of a universal "conductor" manager: a single code-based component responsible for orchestrating agent execution. This approach implements the principle of "One

framework, countless scenarios," where all domain-specific logic is expressed in natural language, while execution is handled by a static, unchanging program. The manager acts as an interpreter of agent swarms, realizing the execution graph, managing states, handling errors, and enabling true parallelism through asynchronous execution. This makes the system suitable for integration into industrial environments requiring reliability, scalability, and determinism.

The scientific novelty of this work lies in the first attempt to formalize no-code programming through mathematical primitives and a structured agent model. We present not only a conceptual framework but also its formal definition, enabling theoretical analysis, automatic prompt generation, and deterministic execution. For the first time, we introduce a formal agent description with a clear separation of role, logic, state, and communication – providing a foundation for developing a theory of LLM-based agent systems. Additionally, we propose a novel mechanism for parallel execution via pseudocode, in which the LLM simulates branch independence through explicit description – an important advancement toward scalable multi-agent systems.

The practical significance of the framework lies in its ability to transform natural language queries into structured, executable instructions that can be integrated into business processes, analytical pipelines, and decision-making systems. The adoption of a standardized output format in structured JSON enables seamless integration of results into other systems, including web interfaces, databases, dashboards, and APIs. This transforms AgentFlow from a mere analytical tool into a platform for building complex intelligent applications accessible to users without technical expertise.

Thus, AgentFlow represents not only a methodological but also an architectural breakthrough in human-AI interaction. It demonstrates that LLMs can serve not merely as text generators, but as interpreters of logical systems capable of executing complex tasks according to well-defined algorithms. This paves the way for the development of scalable, adaptive, and accessible intelligent systems in which humans define the logic, and machines ensure execution.

References

[1] Lande, D. and Strashnoy, L., 2025. AgentFlow-No-Code Agent Framework Based on Logical Primitives. *SSRN Preprint:* 5285664.

[2] LangChain Documentation. *GitHub Repository*. URL: https://docs.langchain.com

[3] Microsoft Semantic Kernel. *GitHub Repository*. URL: https://github.com/microsoft/semantic-kernel

[4] Torronto, G. (2023). AutoGPT: An Experimental Open-Source Attempt to Make GPT-4 Fully Autonomous. *GitHub Repository.* URL: https://github.com/mrgnlabs/Auto-GPT

[5] Singh, Y. (2023). BabyAGI: AI-powered task management system in JavaScript. *GitHub Repository.* URL: https://github.com/ericciarla/babyagijs

[6] Sirui Hong, Mingchen Zhuge, Jiaqi Chen et al. (2023). MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework. *arXiv Preprint:* 2308.00352.

[7] Chen Qian, Wei Liu, Hongzhang Liu et al. (2023). ChatDev: Communicative Agents for Software Development. *arXiv Preprint:* 2302.01486.

[8] Xinyi Li, Sai Wang, Siqi Zeng et al. (2024). A survey on LLM-based multi-agent systems. *Vicinagearth* 1, 9 (2024). DOI: 10.1007/s44336-024-00009-2.

[9] Shanshan Han, Qifan Zhang, Yuhang Yao et al. (2024). LLM multi-agent systems: Challenges and open problems. *arXiv Preprint:* 2402.03578.

[10] Zeyu Zhang, Xiaohe Bo, Chen Ma et al. (2024). A survey on the memory mechanism of LLM-based agents. *arXiv Preprint*: 2404.13501.

[11] Lande, D., & Strashnoy, L. (2025). An Advanced No-Code Programming Framework for Complex Problems in LLM Environments. *ResearchGate Preprint*. DOI: 10.13140/RG.2.2.25307.89129

[12] Dmitry Lande, Leonard Strashnoy. Semantic AI Framework for Prompt Engineering. *SSRN Preprint*: 5172867. DOI: 10.2139/ssrn.5172867

[13] Silva-Aguilera, R.A., Escolero, O., Alcocer, J., Morales-Casique, E., Olea-Olea, S., Vilaclara, G., Lozano-García, S. and Correa-Metrio, A., 2025. Climate and anthropic agents lead to changes in groundwater-surface water interactions in a semi-arid maar lake. Journal of South American Earth Sciences, 156, p.105398.

[14] 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/ *SSRN Preprint*. DOI: 10.2139/ssrn.4978924 (Oct 17, 2024).

[15] Zhou, Z.O.U. and Zhengkang, Z.U.O., 2025. AI Chain-Driven Control Flow Graph Generation for Multiple Programming Language. *Wuhan University Journal of Natural Sciences*, 30(3), pp.222-230. DOI: 10.1051/wujns/2025303222

[16] Pahune, S.; Akhtar, Z. Transitioning from MLOps to LLMOps: Navigating the Unique Challenges of Large Language Models. *Information* 2025, *16*, 87. DOI: 10.3390/info16020087