Extension of the Universal Conductor for Heterogeneous Execution Environments

Dmitry Lande[1], Leonard Strashnoy[2], Viktor Kuzminskyi[3]

[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)

[3] ORCID: 0009-0009-2335-6089 National Technical University of Ukraine – Igor Sikorsky Kyiv Polytechnic Institute

Abstract

Building upon the AgentFlow framework, which introduced a no-code programming paradigm through natural language prompts, this paper addresses the critical challenge of its practical deployment and resource efficiency. AgentFlow features a "conductor" that enables true parallelism and deterministic execution, yet assumes a homogeneous environment in which every agent is executed by a Large Language Model (LLM). We propose a significant evolution of the conductor into an orchestrator with intelligent workload distribution, capable of dynamically assigning agent execution to the most appropriate and cost-effective backend: LLMs for tasks requiring deep reasoning, compact LMs or Small Language Models (SLMs) for moderate tasks, or deterministic code functions for simple, well-defined operations. This extension preserves the core no-code philosophy – where all logic resides exclusively within prompts – while introducing a new layer of execution intelligence that dramatically reduces resource consumption without sacrificing functionality. We present the formal model, architecture, and implementation of this enhanced conductor, demonstrating its ability to optimize the "cost-performance" curve for complex agent swarms.

Keywords: no-code programming, LLM, SLM, agent swarm, orchestrator with intelligent workload distribution, AgentFlow, heterogeneous execution.

Introduction

The AgentFlow framework, introduced in [1], presented a novel approach to no-code development by enabling the definition of complex system logic exclusively through structured natural language prompts, orchestrated by a universal program – the "conductor." Acting as an external execution manager, this conductor resolved the fundamental limitations of LLM-centric architectures by ensuring true parallelism, state management, and system integration, thereby rendering AgentFlow suitable for industrial applications.

However, a new, pragmatic challenge has emerged: resource efficiency. In the original model, every agent – regardless of task complexity – was executed via an LLM API call. This becomes prohibitively expensive for large – scale or high-frequency operations. Many agent tasks – such as data formatting, simple calculations, or invoking a predefined API – do not require the sophisticated (and

costly) reasoning capabilities of an LLM. These tasks can be handled more efficiently using a small language model (SLM) or even a simple deterministic code function.

We propose an extension of the AgentFlow conductor into a Resource-Aware Universal Orchestrator. The core innovation lies in the conductor's ability to intelligently route agent execution to one of three backends:

1. LLM (Large Language Model): For complex, ambiguous, or creative tasks that require deep reasoning and contextual understanding.
2. SLM (Small Language Model): For tasks that require some degree of language comprehension or generation but are more structured and predictable, where a smaller, cheaper, and faster model is sufficient.
3. Code Function: For purely deterministic, algorithmic tasks (e.g., sorting a list, calculating a sum, validating a format), which can be executed with zero LLM/SLM computational cost.

This approach directly addresses the issue of resource consumption, making AgentFlow systems not only reliable and scalable – as demonstrated in [1] – but also economically viable for continuous, real-world operation.

The problem of LLM cost optimization is well recognized in the industry. Common solutions include model distillation, quantization, or caching. However, these are low-level optimizations applied within the LLM execution layer itself.

The proposed approach is fundamentally different and aligns with the high-level architectural philosophy of AgentFlow. Instead of optimizing the model, we optimize the assignment of work. This is analogous to a modern operating system that does not run every process on the most powerful CPU core, but schedules tasks across CPU cores, GPUs, or even specialized accelerators based on their specific computational requirements.

Existing multi-agent frameworks such as LangChain [2] or AutoGen [3] allow tool usage, which can offload tasks to code, but they require users to explicitly code and integrate these tools. This violates the core principle of no-code development. In our extended AgentFlow framework, the decision of how to execute an agent – whether via LLM, SLM, or code – is either inferred by the conductor from the agent's prompt or specified declaratively within the prompt itself, thereby preserving the user's no-code experience.

The motivation is clear: for an agent whose logic is, for example, "Sort the list of numbers in ascending order," invoking an LLM API is wasteful when a Python sorted() function can accomplish the task instantly, perfectly, and at zero cost. The conductor must be intelligent enough to make this distinction autonomously.

Extension of the Formal Conductor Model

In the original formal model, the conductor D was defined as

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

where E (Executor) was a function

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

which always invoked the LLM.

We now redefine the Executor E as heterogeneous:

$$E(a_i, s_i, d_i, \text{backend}) \rightarrow r_i,$$

where

$$\text{backend} \in \{\text{LLM}, \text{SLM}, \text{CODE}\}.$$

The essence of the extension lies in the introduction of a new component: the Backend Router (R). This router is part of the Scheduler S or of the pre-execution stage controlled by E. Its function is:

$$R(a_i) \rightarrow \text{backend}$$

The router R can operate in two modes:

- Declarative mode: The agent's prompt in the JSON definition may include a semantic execution hint, such as "complexity": "low" or "requires_reasoning": false. The router interprets this hint to select the appropriate backend (e.g., CODE for low-complexity, deterministic tasks). This approach preserves the no-code principle by allowing users to describe what the task entails – not how it should be executed technically.
- Inference mode: The router analyzes the agent's logic and role fields to determine the most suitable backend. For example, prompts containing keywords such as "compute", "sort", "verify", or "convert" may be routed to CODE, while prompts such as "analyze sentiment", "generate creative text", or "reason".

This extension is seamless, since it relies on the syntax and metadata of the prompt rather than its domain-specific semantics, thereby preserving the universality of the conductor.

Architecture and Implementation of the Extended Conductor

The architecture described in [1] remains almost unchanged, with the critical improvement introduced in the Executor module (Fig. 1).

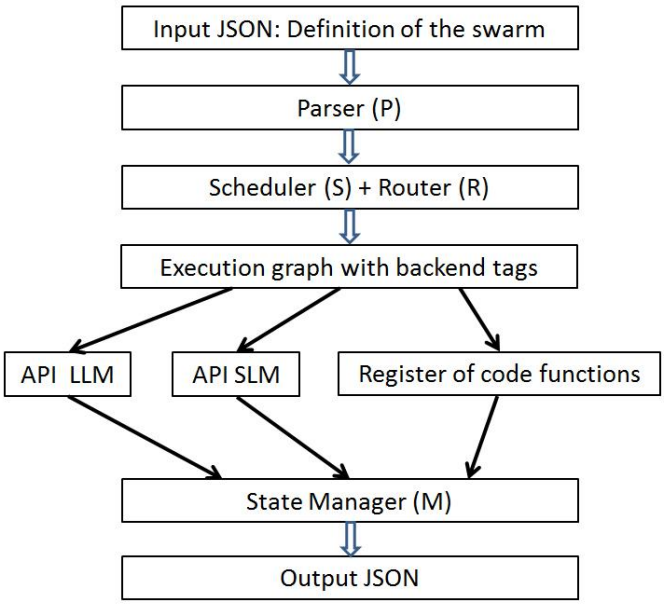

Figure 1. Extended architecture of the conductor with heterogeneous backends

We present the key implementation details:

1. ***Code Function Registry: A Mechanism for Integrating Deterministic Logic into a No-Code System***

In the extended AgentFlow framework, where the orchestrator can execute agents via LLMs, Small Language Models (SLMs), or standard Python functions, a new yet essential component is introduced: the code function registry (code_registry). This registry serves as a centralized library of deterministic procedures that can be invoked in place of an LLM to handle simple, algorithmic, or well-defined tasks described in natural language prompts.

In the original architecture [1], every agent—regardless of task complexity—was executed through an LLM API. This led to three critical inefficiencies:

- Excessive cost: LLM calls incur monetary and latency overhead even for trivial operations (e.g., “Sort a list” or “Check if a number is even”).
- Non-determinism: LLMs may misinterpret unambiguous instructions, reducing reliability.
- Architectural overkill: Using generative AI for tasks with exact algorithmic solutions violates the principle of computational parsimony.

The code_registry resolves these issues by enabling the orchestrator to execute such tasks instantly, deterministically, and at zero LLM cost, using ordinary Python functions. Crucially, the end user—typically a non-programmer—does not need to modify their prompt or even be aware of this mechanism. They simply write, for example, “Sort the list of numbers in ascending order,” and the system automatically routes the task to an appropriate function like sorted().

The registry is implemented as an in-memory dictionary:

```
code_registry = {
   "agent_sort": sort_list,
```

```
    "agent_validate_email": validate_email_format,
    "agent_compute_mean": calculate_mean,
    "agent_capitalize": to_uppercase
}
```

Keys may correspond to agent IDs, roles, or logic hashes; values are references to Python functions.

To maintain full compatibility with the original conductor architecture [1], all registered functions must conform to a strict, unified interface:

```
def func(input_dict: dict, agent_state: dict) -> dict:
    """
    Performs a deterministic operation.
    Args:
        input_dict (dict): Input data for the agent (analogous to 'data' in E(a_i, s_i, d_i)).
        agent_state (dict): Current agent state. May be updated with **universal metadata only**
                            (e.g., invocation counters or timestamps), never domain-specific fields.
    Returns:
        dict: Result formatted identically to LLM/SLM responses.
              Must contain at least: 'output', 'status', and 'error'.
    """
```

This interface ensures three critical properties:

- Input unification: Whether executed via LLM or code, the agent receives data in the same structured format.
- State compatibility: The agent_state is used exclusively for backend-agnostic metadata, such as:
  - invocation_count: total number of executions,
  - last_execution_time: timestamp of the most recent run,
  - error_count: number of failed attempts.
- Domain-specific information (e.g., “number of sorted items”) must never be stored in agent_state, as this would violate the core AgentFlow principle that “all logic resides exclusively in the prompt.”
- Output standardization: All results are returned as JSON-like dictionaries, ensuring seamless integration with the State Manager and Scheduler.

Example implementation:

```
def sort_list(input_dict: dict, agent_state: dict) -> dict:
    """Sorts the list of numbers from the 'numbers' field in the input dictionary."""
    try:
        numbers = input_dict.get('numbers', [])
        if not isinstance(numbers, list):
            raise ValueError("The 'numbers' field must be a list.")
        sorted_numbers = sorted(numbers)

        # Update only universal state fields
        agent_state['invocation_count'] = agent_state.get('invocation_count', 0) + 1

        return {
            "output": sorted_numbers,
            "status": "success",
            "error": None
```

```
        }
    except Exception as e:
        return {
            "output": None,
            "status": "error",
            "error": str(e)
        }
```

Note that no task-specific data is written to agent_state. If downstream agents require semantic details (e.g., "list length"), they are included in the "output" field and referenced via natural language prompts—preserving the no-code paradigm.

The registry is not a standalone service but an integrated part of the Executor (E). It works in concert with:

- Router (R) – decides that an agent should use the CODE backend,
- Executor (E) – invokes the function from code_registry instead of an LLM API,
- State Manager (M) – processes the result identically, regardless of origin.

The registry can be:

- Static: populated at startup with common utility functions,
- Dynamic: extended at runtime via plugins or configuration,
- Automatic: populated by matching prompt keywords (e.g., "sort" → sort_list).

Critically, this does not violate the no-code principle:

- End users never interact with code—they describe tasks in natural language.
- Platform developers implement the function library once, benefiting all users indefinitely.
- New functions can be added without modifying the core conductor logic.

This design mirrors how operating systems abstract hardware: users click "Save," unaware of the underlying filesystem calls. Similarly, AgentFlow users write prompts, unaware of whether execution occurs via LLM, SLM, or deterministic code. Advantages of the code function registry:

CRITERION	LLM-BASED APPROACH	CODE-BASED APPROACH
Cost	High (each call is billed)	Negligible (local execution)
Speed	Slow (API call + generation time)	Instant (local function call)
Reliability	Potential interpretation errors	Deterministic, 100% accurate
Scalability	Limited by cost and latency	High (scales to thousands of concurrent executions)
Energy Efficiency	Low (cloud-based LLMs consume significant energy)	High (local CPU execution is energy-efficient)

Thus, the code function registry is not merely a technical component but a strategic architectural element that enables AgentFlow to evolve from a toy system into an industrial-grade platform. It allows us to:

- Preserve the no-code philosophy for end users.
- Significantly reduce operational costs.
- Increase execution speed and reliability.
- Achieve genuine scalability.

It serves as a bridge between the world of natural language and the world of deterministic code – one carefully engineered so that users never even notice they have crossed it.

2. Executor Logic: Intelligent Selective Execution

In the baseline version of the AgentFlow framework, the Executor operated as a universal interpreter: for each agent, it constructed a textual prompt containing the agent's role, logic, state, and input data, then submitted this prompt to a large language model (LLM) via an API for execution. This approach guaranteed universality – any prompt could be "executed" – but offered no efficiency. Even the simplest operations, such as sorting a list or validating a format, required invoking an expensive LLM, rendering the system impractical for large – scale applications.

Our enhanced Executor retains universality while introducing an intelligent layer for selecting the appropriate execution backend. Rather than always delegating to an LLM, it analyzes the agent's specification and determines the most suitable runtime environment: a full – fledged LLM for complex reasoning tasks, a lightweight small language model (SLM) for moderate workloads, or a standard Python function for deterministic operations. This enhancement leaves the system's external interface unchanged – all results are still returned in JSON format as before – but fundamentally transforms the internal mechanics, making the system highly resource – efficient.

At the core of this mechanism lies a conditional construct – the central element of the executor's logic that determines the execution pathway for each agent. This construct functions as an "execution dispatcher": it receives an agent specification as input, identifies the optimal backend (via the Router component), and then executes the corresponding action. Crucially, this construct contains no domain – specific logic; it operates solely on prompt metadata, adhering strictly to the philosophy that "all logic resides in the prompt."

Below is a conceptual implementation of this conditional construct in Python:

```
def execute_agent(agent, state, data):
    """
    Executes an agent using the most efficient available backend.
    Returns the result in a system – compatible format (JSON – like dictionary).
    """
    # Step 1: Determine the execution backend (LLM, SLM, or CODE)
    # This may be explicitly specified in the prompt or inferred from its content.
```

```
    backend = router.select_backend(agent)  # Invokes the Router component R(a_i)

    # Step 2: Execute the agent according to the selected backend – this is the conditional
construct itself:
    if backend == "CODE":
        # Execute as a standard Python function – fastest and cost – free.
        func = code_registry.get(agent.id)  # e.g., sorted(), sum(), re.match()
        if func is None:
            raise ValueError(f"Function for agent {agent.id} not found in registry.")
        result = func(data, state)  # Call the function with data and state

    elif backend == "SLM":
        # Execute via a small language model API (e.g., Phi – 3, Gemma).
        prompt = build_prompt(agent, state, data)  # Construct textual prompt as in [1]
        result = call_slm_api(prompt)  # Invoke SLM – cheaper than LLM

    else:  # backend == "LLM" (default)
        # Default behavior – as in the original paper [1].
        prompt = build_prompt(agent, state, data)
        result = call_llm_api(prompt)  # Invoke LLM for complex tasks

    # Step 3: Return the result in a standardized format
    return result
```

This seemingly simple if–elif–else construct serves as an architectural bridge between the world of natural language (prompts) and the realm of optimized execution. It enables:

- Preserve the conductor's universality. Regardless of the selected backend, the output format (result) remains consistent. This implies that the State Manager and Scheduler are – and must remain – agnostic to how the result was obtained.
- Implement a function registry. To support the CODE branch, a code_registry must be introduced: a dictionary storing references to Python functions. These functions must adhere to a standardized interface: func(input_dict, agent_state: dict) – > dict, ensuring full compatibility with outputs generated by LLMs/SLMs.
- Enable flexible routing. The router.select_backend(agent) function can be implemented in diverse ways – from interpreting semantic hints in the agent definition (e.g., "complexity": "low", "requires_creativity": false) to performing NLP-based analysis of the agent.logic text. This ensures that even in declarative mode, users never need to reference implementation-specific backends like "LLM" or "CODE". This flexibility allows the system to operate either as explicitly user – controlled or as fully autonomous.
- Maintain the no-code paradigm. End users crafting prompts need not be aware of this conditional construct at all. They merely describe the task, and the system autonomously determines the most efficient execution strategy, remaining a seamless and intuitive tool from the user's perspective.

Thus, this conditional construct is not merely a technical detail but a foundational mechanism that transforms the conductor from a passive prompt executor into an intelligent orchestrator of resources – capable of automatically selecting the optimal tool for each task without compromising the purity of the no-code philosophy.

3. State Manager Compatibility: Preservation of Architectural Integrity

The State Manager (M), as defined in the original formal model of the conductor [1], constitutes one of the four invariant components of the system:

Its function is to store, update, and transmit the internal state of each agent between execution iterations. This state is not merely data but serves as a memory and self-improvement mechanism, enabling agents to adapt based on past experience (e.g., incrementing a call counter, logging common errors, or accumulating accuracy statistics).

In the extended version of the conductor, where agent execution may be delegated to three distinct backends (LLM, SLM, CODE), a key architectural requirement is that the state manager must remain agnostic to the specific backend employed. It must handle any result r_i uniformly, irrespective of whether it originates from GPT-4, a local function such as sorted(), or a small model like Gemma.

Compatibility is ensured at the level of the executor interface (E). Regardless of the chosen backend, the executor returns its result in a strictly defined format – a structured dictionary (dict) containing at least the following fields:

```
{
  "output": ...,          # Main execution result (can be any object: number, string, list, dictionary, etc.)
  "status": "success" | "error",  # Execution status
  "error": str | None,          # Error message (if status == "error")
  # Optional:
  "execution_time": float,       # Execution time (for monitoring purposes)
  "backend_used": "LLM" | "SLM" | "CODE"  # Metadata (for logging only, not for logic!)
}
```

This format is a direct equivalent of the response originally returned by the LLM in version [1]. For instance, when the LLM executed the task "Sort a list," it returned a JSON – like structure. Now, when the same task is performed by the built – in sorted() function via the CODE backend, it mimics this format, returning:

```
{
  "output": [1, 3, 5, 7],
  "status": "success",
  "error": None
}
```

This ensures the following:

- Preservation of modularity. The state manager M implements the function $M(s_i(t),r_i)\rightarrow s_i(t+1)$. It receives the current state $s_i(t)$ and the result r_i, and

updates the state accordingly. If the format of r_i varies depending on the backend, the logic of M must be extended with conditional branches to handle each specific case. This violates the single – responsibility principle and renders the system fragile. A unified format allows M to remain simple, stable, and unchanged.

- Preservation of determinism. As emphasized in [1], one of the key advantages of the conductor is ensuring deterministic execution. If the state manager begins reacting differently to results originating from different backends, the system loses predictability. A unified interface guarantees that M always updates the state identically, irrespective of the data source.
- Transparency for the scheduler (S). The scheduler constructs an execution graph G=(V,E) based on dependencies among agents. It determines, for instance, that agent A must wait for a result from agent B. This mechanism depends solely on the fact that a result has been received and its status (success/error), not on the content of the result itself. A standardized format for ri enables the scheduler to treat any agent as a "black box."
- Support for self – improvement. Reference [1] provides an example where, upon encountering an error, an agent appends it to its state:

  $$s_i.\text{errors.append}(\text{task\_id}, r_i.\text{error}).$$

  This mechanism functions unchanged because the field ri.error exists uniformly for LLMs, SLMs, and CODE backends alike. For instance, if a function such as validate_email detects an invalid format, it returns:

```
{
   "output": None,
   "status": "error",
   "error": "Invalid email format: 'user@'"
}
```

The state manager handles this in exactly the same way as an error originating from the LLM.

Architectural Insight: Encapsulation of Complexity

This implementation serves as a clear illustration of the encapsulation principle. All complexity associated with backend selection, API invocation, function execution, or exception handling is fully encapsulated within the executor (E). To external components – the scheduler (S) and the state manager (M) – the executor remains a "black box" with a simple interface:

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

Consequently:

- Modularity is preserved. Each component performs a clearly defined function.
- Scalability is maintained. Adding a new backend (e.g., an “ML_MODEL” backend for executing sklearn models) requires no modifications to S or M; it suffices to extend the logic within E.

- Stability is ensured. The architectural core (P, S, M) remains unchanged, thereby upholding the philosophy of "One platform – many scenarios."
- Result unification as an architectural guarantee. Thus, standardizing the result format r_i is not merely a technical detail but an architectural guarantee that extending the conductor's functionality will not compromise its core. This enables the execution layer to evolve without undermining the foundation established in [1].

Example Scenario

Consider the complex scenario from [1], which involved data preprocessing, time – series analysis, text analysis, and report generation.

- Agent 1 (Data Preparation): Logic: "Clean the data and transform it into a structured format." This typically involves removing null values, standardizing formats, etc. This task can be ideally handled by the CODE backend using predefined data – cleaning functions.
- Agent 2 (Time Series Analysis): Logic: "Compute trend and seasonality." This is a well – defined statistical task and can be efficiently delegated to the CODE backend using libraries such as statsmodels.
- Agent 3 (Text Analysis): Logic: "Determine the sentiment of a news article." This requires nuanced language understanding and is best suited for an LLM (or possibly an SLM, if the sentiment classification task is straightforward).
- Agent 4 (Report Generation): Logic: "Integrate all results into a final conclusion." This task involves synthesizing information and generating coherent narrative text, making it ideally suited for an LLM.

Execution via an extended orchestrator:

1. Router R analyzes each agent and assigns backends: Agent 1 → CODE, Agent 2 → CODE, Agent 3 → LLM, Agent 4 → LLM.
2. Scheduler S constructs the graph as before.
3. Executor E runs Agents 1 and 2 using inexpensive, fast code functions.
4. Executor E executes Agents 3 and 4 in parallel via the LLM API.
5. State Manager M updates states in a unified manner.

For instance, an agent for data cleaning might be defined as:

```
{
  "id": "agent_1",
  "role": "Data Preprocessor",
  "logic": "Clean the data and transform it into a structured format.",
  "complexity": "low",
  "requires_reasoning": false
}
```

The router interprets "complexity": "low" and "requires_reasoning": false as signals to route execution to the CODE backend.

Result: The overall cost and execution time of the swarm are significantly reduced, since two of the four agents no longer incur any LLM API expenses, while the

system's output and functionality remain identical to the original, fully LLM – based implementation.

Conclusions

The scientific contribution of this work is the extension of the formal model of no-code orchestration to support heterogeneous execution environments. We introduce the concept of a backend-agnostic execution layer within the universal orchestrator, where the execution method is decoupled from the execution objective defined in the prompt. This establishes a new dimension of optimization for agent – oriented systems: not only what to execute, but also where to execute it.

The practical contribution is substantial. It makes the AgentFlow framework suitable for production environments by directly addressing the primary adoption barrier: cost. Non – technical users can now design complex workflows with the assurance that the system automatically minimizes resource consumption. Organizations can define execution policies (e.g., "attempt CODE first, then SLM, and use LLM only if necessary"), which the orchestrator enforces universally, resulting in significant operational cost reductions.

The AgentFlow framework has previously introduced a no-code solution for orchestrating intelligent agents with deterministic, parallel execution. This article advances that foundation by presenting the Resource-Aware Universal Orchestrator – an extended orchestrator capable of routing agent execution across LLMs, SLMs, or deterministic code functions.

Importantly, even in declarative configuration, the framework avoids exposing technical backend names (e.g., "LLM", "CODE") to the user. Instead, it relies on semantic task descriptors such as complexity level or reasoning requirements, ensuring that the no-code experience remains intuitive and implementation-agnostic.

This development represents not a departure from, but a maturation of, the original design philosophy. By aligning the computational backend with the complexity of the task, we preserve the accessibility and expressive power of no-code prompt engineering while ensuring economic sustainability. The orchestrator thus functions not only as an execution manager but also as an optimization component that guarantees appropriate resource selection for each task.

This enables large-scale, cost-effective deployment of AgentFlow-based systems in business, scientific, and industrial contexts, where efficiency and scalability are critical requirements.

References

[1] Lande, D. and Strashnoy, L., 2025. Orchestration of No-Code Agents in the AgentFlow Framework. SSRN Preprint: 5381820.

[2] Topsakal, O. and Akinci, T.C., 2023, July. Creating large language model applications utilizing langchain: A primer on developing llm apps fast.

In International conference on applied engineering and natural sciences (Vol. 1, No. 1, pp. 1050 – 1056).

[3] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J. and Awadallah, A.H., 2024, August. Autogen: Enabling next – gen LLM applications via multi-agent conversations. In First Conference on Language Modeling.