AutoStreamPipe: LLM Assisted Automatic Generation of Data Stream Processing Pipelines

Abolfazl Younesi[1,*], Zahra Najafabadi Samani[1,*], Thomas Fahringer[1,*]

Departement of Computer Science, University of Innsbruck, Technikerstraße, Innsbruck, 6020, Tirol, Austria

Abstract

Data pipelines are essential in stream processing as they enable the efficient collection, processing, and delivery of real-time data, supporting rapid data analysis. In this paper, we present AutoStreamPipe, a novel framework that employs Large Language Models (LLMs) to automate the design, generation, and deployment of stream processing pipelines. AutoStreamPipe bridges the semantic gap between high-level user intent and platform-specific implementations across distributed stream processing systems for structured multi-agent reasoning by integrating a Hypergraph of Thoughts (HGoT) as an extended version of GoT. AutoStreamPipe combines resilient execution strategies, advanced query analysis, and HGoT to deliver pipelines with good accuracy. Experimental evaluations on diverse pipelines demonstrate that AutoStreamPipe significantly reduces development time (×6.3) and error rates (×5.19), as measured by a novel Error-Free Score (EFS), compared to LLM code-generation methods.

Keywords: Large Language Models, Stream Processing, Data Pipelines, Workflow Automation, Hypergraph of Thoughts, Multi-Agent Systems, Pipeline Generation

1. Introduction

The rapid evolution of technology has made stream data processing essential rather than optional [1, 2]. Stream processing (SP) pipelines form the backbone of systems that require low latency and high throughput for data. These systems range from monitoring IoT devices and sensor networks in cyber-physical systems to detecting fraudulent financial transactions [3, 4, 5]. To meet these demands, SP pipelines must transform raw data streams into actionable insights in real time. However, their development remains complex and time-consuming due to the intricacies of distributed, stateful computation.

Challenges in Pipeline Development. Designing and deploying SP pipelines presents significant challenges. Traditional approaches typically involve manual coding, iterative debugging, and labor-intensive optimization [6].

Developing data pipelines demands expertise in two key areas: domain-specific logic (e.g., rules for anomaly detection) and framework-specific APIs (e.g., Flink's DataStream API), which poses a significant challenge for domain experts, such as data analysts, who may lack advanced programming skills. As a result, manual coding by engineers often involves considerable effort refining logic, debugging edge cases, and optimizing resource usage. Even automated code generation tools fall short, as they fail to bridge the semantic gap between high-level intent (e.g., "detect three consecutive failed logins within five minutes") and low-level implementation details (e.g., configuring

Flink operators with custom triggers). This gap highlights the need for a paradigm that combines the accessibility of low-code interfaces with the expressivity of handcrafted code.

Although automated code generation and template-driven tools exist [7, 8], they often fail to address dynamic operations, particularly stateful operations [6]. While effective for building pipelines with simple, common operators, they typically require users to write custom code for complex or domain-specific logic, reintroducing the manual effort they were designed to avoid.

To address these challenges, several stream processing application benchmarks have been proposed. However, existing benchmarks still exhibit significant limitations[9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22]. Benchmarks such as RIoTBench [11], DSPBench [12], HiBench [14], BigDataBench [19], the Linear Road Benchmark [15], NexMark [9], BigBench [20], and LinkBench [21] often rely on outdated DSPS versions, support only a limited range of platforms, and fail to incorporate recent advancements in state management and event-time processing. As a result, their relevance to modern deployments is significantly constrained. As shown in Table 1, existing benchmarks rarely provide comprehensive cross-platform evaluations. Additionally, they typically offer a narrow set of predefined pipelines, many of which are synthetic and fail to reflect real-world scenarios. These gaps in evaluation capabilities motivate our work. Rather than proposing another static benchmark suite, we focus on automating the generation of diverse SP pipelines, providing a flexible source of workloads that complements existing benchmarking efforts.

Introducing LLMs for Pipeline Generation. Recent ad-

---

*Corresponding authors

Email addresses: abolfazl.younesi@uibk.ac.at (Abolfazl Younesi), zahra.najafabadisamani@uibk.ac.at (Zahra Najafabadi Samani), thomas.fahringer@uibk.ac.at (Thomas Fahringer)

vances in automation, driven by the rise of LLMs such as GPT-4 [23], LLaMA [24], and Code Llama [25], offer promising avenues to address pipeline development challenges. LLMs can understand natural language descriptions, generate code, and reason through complex logic. This could make advanced technologies, such as distributed stream processing systems like Apache Flink [26, 27], Apache Storm [28, 29], Kafka Streams [30], and Apache Spark Streaming [31], more accessible.

Despite their potential, LLMs face significant limitations when applied to pipeline generation, including challenges with semantic accuracy, inconsistency in multi-step reasoning, and difficulty adapting to evolving requirements, such as changes in data schemas or real-time processing constraints. **Proposed Approach.** To address the challenge of rapid pipeline prototyping, we propose AUTOSTREAMPIPE, a framework that employs LLMs [32] to automate SP pipeline generation (cf. Figure 1 ①-⑤). AUTOSTREAMPIPE includes a query analyzer integrated with the plan designer and short-term and long-term memory for future references. We embed a Hypergraph of Thought (HGoT) and integrate it with multiple LLM agents in a RAG setup, based on standard SP pipeline architectures, to produce production-ready pipelines. In contrast to general-purpose code generators that often produce unclear or non-optimizable execution plans using outdated or deprecated functions, we automate the creation of clean, high-level code. This approach ensures the framework serves as a solid starting point for quick validation and iteration with SP technologies. Our framework is capable of generating a virtually unlimited number of SP pipelines, supporting both synthetic and real-world use cases across all major DSPSs. This paradigm accelerates development cycles and lowers the technical barrier to using SP technologies, enabling domain experts without programming expertise to translate their specialized knowledge into operational systems.

Contribution. Our main contributions are as follows:

• **End-to-End Automation:** Our system automates the entire pipeline lifecycle, from interpreting user inputs to generating, optimizing, and validating the final pipeline for DSPS based on high-level semantic understanding (cf. Section 4).

• **Query Analyzer**: We introduce a dedicated module for intent detection and parameter extraction that deconstructs natural language queries into formal pipeline specifications. It performs deep semantic analysis to infer implicit constraints, validate explicit user constraints, and generate execution plans tailored to varying pipeline complexities (cf. Section 4.2).

• **Hypergraph of Thoughts (HGoT) Reasoning:** We propose HGoT, a novel structured reasoning framework that extends the Graph of Thoughts (GoT) paradigm [33] by introducing hyperedges to model multi-way dependencies among partial solutions. HGoT enables coordinated reasoning across interdependent steps, improving consistency and efficiency in complex pipeline synthesis tasks (cf. Section 4.3).

• **Resilient Multi-Agent Execution Infrastructure:** We implement a fault-tolerant multi-agent architecture that ensures

Table 1: Comparison among widely used benchmark suites for data stream processing pipelines (continuous queries). † means a framework that can generate infinite benchmark pipelines

Benchmark Suite	Real-world App.	Synthetic App.	DSPSs	Unified API	Workload Charact.
Linear Road Benchmark [15]	1	-	Aurora	No	No
Yahoo Streaming Benchmark [16]	1	-	Storm, Flink, Spark Streaming	No	No
BigDataBench [19]	-	1	Spark Streaming	No	No
StreamBench [18]	-	7	Storm, Spark Streaming	No	Yes
RIoTBench [11]	4	-	Storm	Yes	Yes
HiBench [14]	-	7	Storm, Flink, Spark Streaming	No	No
DSPBench [12]	13	2	Storm, Spark Streaming	Yes	Yes
AUTOSTREAMPIPE†	7	1	Storm, Spark, Flink	Yes	Yes

reliability. It intelligently rotates between different LLMs, retries failed tasks, and combines specialized models to overcome individual API errors or performance issues, guaranteeing a robust generation process (cf. Section 4.4).

• **Open-Source Prototype and Evaluation Metric:** We release an open-source implementation of our system, supporting a wide range of DSPS frameworks and reducing pipeline development time by up to 6.3× in experimental evaluations (cf. Section 5). Additionally, we introduce the *Error-Free Score (EFS)*, a novel metric for quantitatively assessing the correctness and completeness of LLM-generated pipelines, offering a more rigorous and holistic evaluation framework than existing measures (cf. Section 5.3).

Paper organization. The remainder of this paper is organized as follows: Section 3 explains background concepts in SP. Section 4 describes the AUTOSTREAMPIPE architecture in detail. Section 5 presents our comprehensive evaluation of AUTOSTREAMPIPE. Section 2 reviews related work. Section 6 concludes the paper.

2. Related Work

The increasing complexity of DSP pipelines has driven significant research and development efforts toward automating their design and optimization [34, 35]. These efforts aim to reduce the manual effort required for pipeline creation, improve
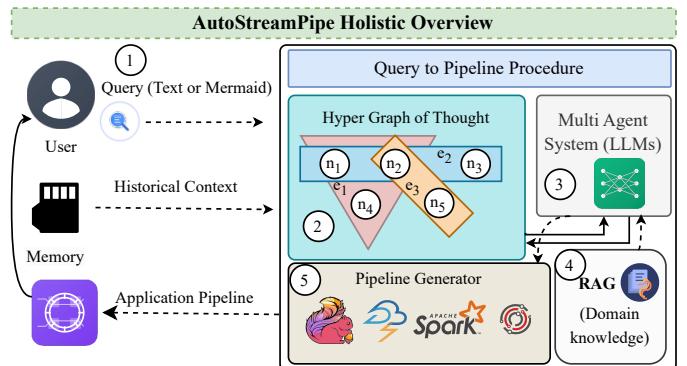

Figure 1: A holistic overview of the AUTOSTREAMPIPE framework

performance, and make SP more accessible to non-experts. Machine learning (ML)-based tools, for example, employ reinforcement learning, supervised learning, and neural architecture search to generate workflows or ML pipelines similar to data pipelines and query execution [36, 37, 38, 39]. However, these techniques typically demand extensive training data and computational power.

Low-code and no-code platforms [40] such as Apache NiFi [41, 42, 43], StreamSets [44], and Node-RED [45] offer graphical interfaces that simplify pipeline creation but may lack the flexibility and performance needed for complex or high-throughput scenarios. Rule-based systems, found in many query optimizers within engines like Apache Flink and Spark Streaming, as well as in tools like StreamMine3G [46], utilize predefined heuristics to enhance efficiency but struggle with ambiguous requirements. Template-based solutions offer reusable patterns through cloud services such as Google Cloud Dataflow [47] and AWS Kinesis, as well as open-source libraries like Streamdrill and Siddhi, although they often fall short in addressing unique needs. Finally, hybrid approaches combine declarative programming models like Flink SQL or KSQL with AI-augmented design tools, offering a balance between ease of use and powerful optimization while requiring advanced infrastructure and expertise.

Recent advances in LLMs have demonstrated remarkable capabilities in automated code generation and software engineering tasks [48, 49, 50]. Foundation models such as CodeLlama [25], StarCoder [51], and CodeGen [52] have been specifically trained on massive code corpora, achieving strong performance across multiple programming languages and paradigms. Transformer-based approaches, such as AlphaCode [53], have demonstrated competitive performance in programming competitions. In contrast, CodeT5 [54] utilizes encoder-decoder architectures for code understanding and generation tasks. Commercial tools, such as GitHub Copilot and Amazon CodeWhisperer, have introduced AI-assisted coding into mainstream development workflows, demonstrating the practical viability of LLM-based code generation. However, these general-purpose code generation models face significant challenges when applied to domain-specific contexts, such as stream processing pipelines. They do not adequately consider streaming-specific constraints, such as stateful operations, windowing semantics, checkpointing requirements, and fault-tolerance guarantees. As a result, the generated code often suffers from incorrect state handling, data loss, or inconsistent event ordering, making it unreliable for real-time execution in DSP environments. Moreover, they typically generate isolated code snippets rather than complete, production-ready pipeline architectures with proper configuration, deployment specifications, and operational considerations. AUTOSTREAMPIPE addresses these limitations by combining LLM capabilities with domain-specific reasoning frameworks (HGoT), retrieval-augmented generation for streaming domain knowledge, and multi-agent collaboration to ensure coherent and executable pipeline solutions that satisfy the complex interdependencies inherent in DSPS.

On the other hand, there are some papers published recently that use LLMs to generate workflows for serverless computing [55, 56, 57, 58]. These papers generate YAML files and are different from our purpose.

Summary. Automated SP pipeline design has evolved significantly through several complementary approaches, including machine learning optimization, low-code/no-code platforms, rule-based systems, template-based solutions, and, more recently, LLM-based code generation. Each approach has its own benefits. ML methods are effective for performance optimization, but they require a substantial amount of training data. Low-code platforms make things easier to access, but lose some flexibility. Rule-based systems offer clear optimization but struggle with unclear requirements. Template-based solutions enable quick deployment but often lack customization options. General-purpose LLM code generators demonstrate impressive capabilities but lack understanding of specific streaming requirements, such as stateful operations, windowing rules, checkpointing intervals, and fault tolerance guarantees. Despite these advancements, current methods often fail to address the various challenges that modern SP systems encounter fully. There has been no effort to combine natural language understanding, domain-specific reasoning for complex streaming connections, multi-agent teamwork for reliable execution, and complete generation of production-ready pipelines with all necessary configuration and deployment details. This gap drives the need for LLM-assisted approaches that mix domain knowledge, structured reasoning frameworks, and strong execution strategies. This integration aims to connect human intent with machine execution, enabling the generation of smart, fully automated pipelines that meet both functional needs and operational constraints.

3. Background

Over the years, many DSPS have emerged to handle continuous data flows in real-time [59, 60, 2, 6], including Apache Flink [61], Storm [62], Spark Streaming [63], and Kafka Streaming [64]. Below, we provide an overview of key components and stages of a typical DSP.

3.1. Data Stream Pipeline (DSP)

Definition. A DSP is an end-to-end system designed to process continuous data streams in real-time (cf. Figure 2) [2]. It consists of interconnected operators that perform tasks such as data ingestion, transformation, and output delivery [2, 59, 60].

Operators are the fundamental building blocks of DSPs, enabling modular, scalable, and flexible data processing. By chaining operators, DSPs move data seamlessly from sources (e.g., sensors, logs) to destinations (e.g., databases, APIs) while applying necessary computations and transformations.

Data Ingestion. Source operators capture raw data from sources such as Kafka topics, files, or sensors and feed it into the pipeline. These operators serve as the starting point for all downstream processing.

Data Transformation. This core stage involves cleaning, filtering, aggregating, and enriching data to make it actionable. Transformations can be stateless (e.g., filtering, mapping) or stateful (e.g., windowing, joins) [65].
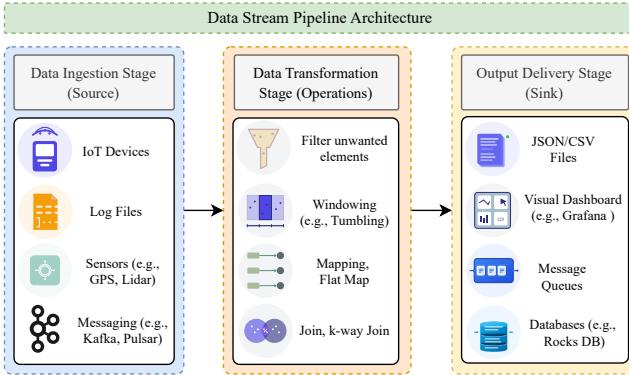
Figure 2: An overview of a data stream pipeline structure.

Output Delivery. Sink operators store or deliver processed data to external systems such as databases, file systems, or dashboards, ensuring the results are accessible and actionable.

Despite their strengths, DSPSs require significant manual effort to design, configure, and optimize pipelines. This challenge motivates our exploration of LLM-assisted approaches to streamline pipeline generation and enhance usability.

4. AutoStreamPipe Architecture

In this section, we introduce AUTOSTREAMPIPE, a framework that automatically translates natural language queries into SP pipelines. The system utilizes multiple LLMs to ensure robustness against common API limitations such as rate limits, outages, or model unavailability. To illustrate how AUTOSTREAMPIPE works, we use a running example introduced in box 1. Each phase of the pipeline generation process is explained by showing how the system transforms this example step by step, from initial query to executable SP pipeline. The overall architecture is depicted in Figure 3, with detailed algorithms provided in Algorithms 1 to 4.

4.1. Preprocessing Stage

The system begins by configuring key parameters, such as maximum file size, chunk size, supported file extensions, and SPE settings (Algorithm 2, line 2). These settings ensure efficient resource utilization and allow for customization without requiring code modifications. This entire stage executes once during system initialization, thereby creating a stable foundation for subsequent operations. This approach addresses the primary limitation of static benchmarks, as outlined in Section 1, which was their inability to keep pace with rapidly evolving SPE frameworks. AutoStreamPipe implements a dynamic repository management system.

Next, the system identifies and clones target GitHub repositories containing the latest pipeline examples and documentation. By default, it targets Apache Flink, Storm, and Spark, though users can specify alternative frameworks or specific versions via a configuration file. This dynamic approach ensures that AutoStreamPipe always has access to current best practices, emerging patterns, and updated APIs from official SPE repositories. The repository cloning mechanism implements robust fault-tolerance strategies to ensure reliable acquisition of

---

Algorithm 1: Execute Step With Retry

```
    Input  : executor, step, plan
    Output: Result of the executed step
1   Function ESR(executor, step, plan):
2       retries, maxRetries ← 0,5;
3       while retries < maxRetries do
4           try:
5               result ← EXECUTESTEP(step.action, plan.query,
                    step.dependencies);
6               return result;
7           catch:
8               APIError e
9           if e.isRateLimit then
10              delay ← baseDelay ×2^retries × (0.5 + random());
11              SLEEP(delay);
12          else if e.isQuotaExceeded then
13              SWITCHTONEXTMODEL(executor);
14          retries ← retries + 1;
15      return GENERATEFALLBACKRESULT(step);
```

remote resources. When network issues or rate limiting occur, the system employs *exponential backoff with jitter*, a proven resilience technique where retry delays increase exponentially (e.g., 1s, 2s, 4s, 8s) while incorporating randomization to prevent synchronized retry storms across multiple instances. This approach, formalized as delay = baseDelay × 2^{retries} × (0.5 + random()) in Algorithm 1 (lines 9-11), significantly enhances system reliability by gracefully handling transient failures without overwhelming external services. Once repositories are cloned, they are organized into a structured local hierarchy for easy access and management. If RAG is enabled, the system scans for pipeline documentation, identifies directories with pipeline templates, and indexes components such as source (SO), operator (OP), and sink (SI) (Algorithm 2). A recursive search efficiently locates directories containing example pipelines, such as the Word Count pipeline. Finally, robust error handling ensures that issues with individual files don't disrupt the workflow. The output includes annotated code examples with metadata and SHA-256 checksums, balancing readability for humans with machine processability while preserving important context.

4.1.1. Error Handling and Output Preparation

Robust error handling is integral to this phase, ensuring that issues with individual files do not disrupt the overall workflow. The system implements graceful degradation strategies that log and skip corrupted or inaccessible files rather than terminating the process. File integrity is verified through SHA-256 checksums, which detect any corruption during transfer or storage. When processing errors occur, the system continues with available resources while maintaining a comprehensive error log for diagnostic purposes.

The output of this phase includes annotated code examples enriched with metadata such as SPE version, component type, and usage context. This dual representation balances human readability with machine processability, preserving important contextual information that guides subsequent pipeline generation steps. By establishing this comprehensive, dynamically updated knowledge base, Phase 1 creates the foundation upon which all subsequent AutoPipe operations build, effectively solving the benchmark obsolescence problem through continuous integration of evolving SPE documentation.
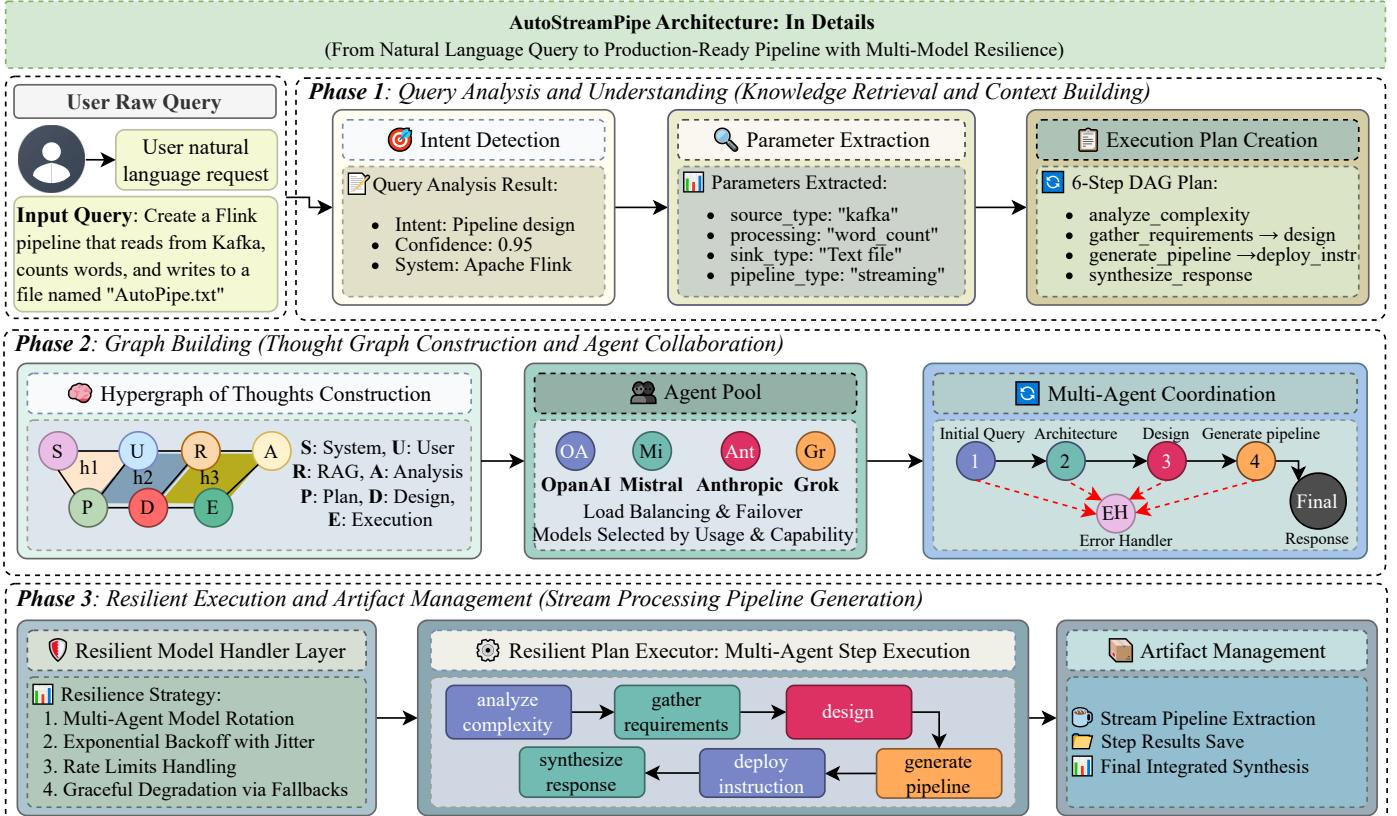
Figure 3: The AUTOSTREAMPIPE architecture presents a novel approach to generating production-ready SP pipelines from natural language queries. The system employs a three-phase methodology: (1) query analysis and parameter extraction, (2) hypergraph-based reasoning with multi-agent collaboration, and (3) resilient execution with comprehensive artifact management. This framework harnesses the capabilities of multiple LLM providers while ensuring resilience against API limitations through retry mechanisms and model rotation strategies.

4.2. Phase 1: Query Analysis and Understanding

The first phase of AUTOSTREAMPIPE processes natural language inputs through a series of analytical steps, as outlined in Algorithm 2.

Step 1: Query Analysis. Intent Detection begins by interpreting the user's request through intent understanding, a two-stage process combining efficiency and adaptability. First, it applies fast regular expression pattern matching (e.g., detecting phrases like in the box ① to infer an `intent type` of `pipeline design`). For ambiguous or complex queries, an LLM-powered intent detector is reconstructed, which structures the input into a lightweight `QueryIntent` object containing: **(1)** a high-level category (e.g., `Pipeline design`), **(2)** a confidence score (a float between 0 and 1), and **(3)** extracted parameters (e.g., framework = "Apache Flink", `window_size` = 5 min).

Step 2: Parameter Extraction. This builds upon the detected intent to identify specific requirements embedded in the query. Based on the intent type (Algorithm 2, line 9), the system employs specialized extraction using an LLM that targets parameters relevant to that particular intent. For pipeline design intents, these parameters include the data source type ("kafka" in the example), processing operations ("word_count"), sink type ("Text file"), and pipeline type ("streaming"). The parameter extraction process employs similar resilient retry capabilities

to intent detection, ensuring robustness when interfacing with external LLM providers. The extraction process balances precision with flexibility, identifying explicit parameters while inferring implicit ones based on domain knowledge of SP systems. The resulting parameter set formalizes the user's requirements, transforming ambiguous natural language into structured data to drive pipeline generation.

Step 3: Create Execution Plan. This step concludes phase 2 by constructing a directed acyclic graph (DAG) that outlines the sequential steps needed to generate the requested pipeline (Algorithm 2, line 10).

Based on the identified intent and extracted parameters, the system customizes the plan with steps suited for the request type, typically including: `analyze complexity` to assess the pipeline's computational characteristics, `gather requirements` to formalize specifications, design to develop the architectural structure, `generate pipeline` to create implementation code, `deploy instructions` to provide operational guidance, and `synthesize response` to produce the final output. The plan establishes dependencies between steps to ensure a coherent execution where each stage builds on its predecessor, offering a clear roadmap for progress. The execution plan converts user intent into a concrete procedure for generating the desired pipeline.

The second phase of AUTOSTREAMPIPE converts unstructured

Algorithm 2: AutoStreamPipe

Input : User query, Streaming system selection, Config options
Output: Stream processing pipeline solution with code and documentation

1 **Function** AutoStreamPipe(*query, streamingSystem, options*):
2   models ← INITIALIZEMODELS(options.modelsList, options.backupModels);
3   planningModel ← GETPLANNINGMODEL(models[0]);
4   retryHandler ← IRH(models);
5   intent ← DETECTINTENTWITHRETRY(retryHandler, query);
6   parameters ← EPR(retryHandler, query, intent);
7   executionPlan ← CEP(query, intent, streamingSystem, options.useRAG);
8   thoughtGraph ← HGoT CONSTRUCTION(streamingSystem, query, options);
9   **if** *options.useRAG* **then**
10   ragDocs ← RRD(query, streamingSystem);
11   **foreach** *doc in ragDocs* **do**
12   ADDNODETOGRAPH(thoughtGraph, doc, "rag");
13   NODETOGRAPH(thoughtGraph, "QueryAnalysisResult", "analysis");
14   NODETOGRAPH(thoughtGraph, "ExecutionPlan", "plan");
15   planExecutor ← CREATERESILIENTEXECUTOR(models, thoughtGraph, streamingSystem);
16   **foreach** *step in executionPlan.steps* **do**
17   **while** *step has unmet dependencies* **do**
18   **continue**;
19   result ← ESR(planExecutor, step, executionPlan);
20   SAVESTEPRESULT(result, step);
21   **if** *result contains JavaCode* **then**
22   javaFile ← EXTRACTANDSAVEJAVACODE(result);
23   MARKSTEPCOMPLETED(step, result);
24   finalResponse ← SYNTHESIZERESPONSE(executionPlan, thoughtGraph);
25   summary ← CREATESESSIONSUMMARY(query, intent, streaming system);
26   SAVEMEMORY(query, finalResponse, thoughtGraph);
27   **return** (*finalResponse, summary*);

---

natural language into structured representations to guide the generation of the pipeline. The process begins when the user submits a request as raw text input to Algorithm 2, as shown in the box 1. The system applies preliminary preprocessing, normalizes the text, removes extra whitespace, standardizes punctuation, and recognizes key entities to formalize implicit requirements and structure the pipeline design.

4.3. Phase 2: Graph Building and Agent Collaboration

The second phase establishes the reasoning framework and agent coordination infrastructure that enable the generation of sophisticated pipelines. In the context of automated pipeline generation, *reasoning* refers to the systematic process of deriving conclusions, making design decisions, and solving problems through logical inference and the integration of knowledge. Specifically, AutoStreamPipe must reason about interdependent components: determining compatible data sources, selecting appropriate operators, configuring state management, and ensuring fault tolerance. All while maintaining consistency across these interconnected decisions.

Pipeline synthesis for streaming data platforms (*e.g.*, Apache Flink) involves interdependent design decisions, including data source configuration, operator chaining, state management, and fault tolerance. For instance, choosing a Kafka source influences the selection of serialization format, which in turn affects the implementation of operators, which in turn constrains sink configuration. These multi-way dependencies create a reasoning challenge: a decision in one component can simultaneously invalidate or necessitate changes in multiple other components. Pairwise graph structures for reasoning like CoT, GoT, where edges connect only two nodes at a time, struggle to encode such multifactor constraints. They force the system into repeated backtracking when dependencies are violated, leading to inconsistent partial plans and computational inefficiency. By con-

---

Word Count Pipeline (Complete Example Version)

Create an Apache Flink streaming application that processes text data with the following specifications:

- **Source:** Kafka topic "input-text" (bootstrap servers: `localhost:9092`, consumer group: `word-count-group`)

- **Input format:** Plain text messages with UTF-8 encoding

- **Processing:** Split messages by whitespace regex `"\s+"`, convert to lowercase, and filter words with length ≥ 3

- **Windowing:** 30-second tumbling windows for aggregation

- **Output:** Local file system at `/output/word-counts.txt` with format "word,count,timestamp"

- **Parallelism:** 4 for source, 8 for processing, 2 for sink

- **Checkpointing:** Every 10 seconds with sqlite3 state backend

- **Error handling:** Dead letter queue for malformed messages to Kafka topic "dlq-text"

Box 1: Word Count Pipeline (Complete Example Version)

trast, HGoT's hyperedges enable the system to bind *sets* of related decisions into coherent reasoning units, facilitating global consistency checks and synchronized updates across multiple dependent components.

Step 1: Hypergraph of Thoughts Construction. The reasoning framework at the heart of AUTOSTREAMPIPE employs a novel cognitive architecture known as the *Hypergraph of Thoughts (HGoT)*. To understand HGoT's advantages, we first examine the evolution of reasoning frameworks in LLMs (see Table 2). Recent reasoning frameworks, such as Chain of Thought (CoT) [66], Multiple Chains of Thought (CoT-SC) [67], Tree of Thoughts (ToT) [68], Graph of Thoughts (GoT) [33], and Layer of Thoughts (LoT) [69], incrementally improve flexibility, complexity, and interpretability. However, these approaches share a fundamental limitation: they represent reasoning structures using pairwise connections among nodes or layers, as illustrated in Table 2 and Figure 4. This pairwise constraint creates a representational bottleneck when modeling real-world problems with multi-way dependencies. Consider pipeline state management: the choice of state backend (e.g., RocksDB vs. heap-based) simultaneously affects checkpointing strategy, recovery time objectives, memory configuration, and operator parallelism. Representing this four-way dependency using pairwise edges requires creating multiple redundant connections and complex coordination logic, obscuring the fundamental unity of these interconnected decisions.

To overcome these limitations, AUTOSTREAMPIPE implements the HGoT architecture, which extends traditional graphs by introducing hyperedges that can simultaneously connect multiple nodes. This hypergraph structure offers a more expressive representation of complex, multifactor constraints and relationships in practical applications, such as scheduling with interdependent tasks.

In the Hypergraph of Thoughts, the nodes represent individual reasoning steps, hypotheses, or partial solutions. Hyperedges link groups of nodes to capture multi-way relationships and interdependencies directly. This structure enables higher-order reasoning and synchronized iterative refinement,

Table 2: Comparison of Advanced Reasoning Frameworks

Feature	Chain of Thought	CoT-SC	Tree of Thoughts	Layer of Thoughts	Graph of Thoughts	Hypergraph of Thoughts
Structure	Linear sequence	Parallel linear sequences	Hierarchical tree	Layered graph (DAG)	Directed graph	Hypergraph
Node Connections	One-to-one (sequential)	One-to-one (within chains)	One-to-many (branching)	Many-to-many (layer-to-layer)	Many-to-many (pairwise)	Many-to-many (group-based)
Reasoning Flow	Unidirectional	Parallel unidirectional	Hierarchical	Layered, feed-forward	Networked	Higher-order networked
Backtracking	Limited/None	None (selection by voting)	Natural	via refinement layers	Supported	group backtracking
Parallelism	Limited	Very high	Moderate	High (within layers)	High	Very high
Multi-constraint	Poor	Moderate	Moderate	Good	Good	Excellent
Group Inference	Not supported	Limited (via aggregation)	Limited	Supported	Limited	Native support
Complexity	$O(n)$	$O(k \cdot n)$	$O(b^d)$	Layer-dependent	$O(n^2)$	$O(2^n)$ worst case
Suitable Problems	Sequential reasoning	Robustness via diversity	Hierarchical decomposition	Layered refinement	Networked dependencies	Complex constraint satisfaction

Note: n: number of reasoning steps/nodes, k: number of chains in CoT-SC, b: branching factor in ToT, d: depth of the tree in ToT
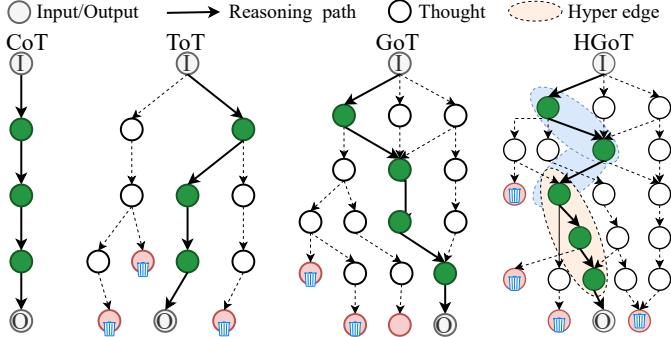

Figure 4: Illustration of the CoT, ToT, GoT, and HGoT reasoning process pipeline. This structure enables higher-order dependencies, dynamic pruning of infeasible paths (represented by red trash-bin icons), and adaptive traversal toward an optimal solution.

ensuring consistency across interconnected reasoning elements. The system iteratively evaluates hyperedge constraints, simultaneously adjusting the involved nodes by refining solutions, merging compatible ideas, or pruning infeasible options. HGoT improves AUTOSTREAMPIPE's ability to manage complexity and converge toward globally coherent solutions by enabling synchronized updates and capturing intricate interdependencies among reasoning steps. This capability supports flexible answer aggregation, accommodating single/multiple outcomes depending on task requirements.

Formal definition. We model reasoning in AUTOSTREAMPIPE as an HGoT, formally defined by the tuple $H = (V, E, \Phi, \Psi)$, where:

- $V = \{v_1, \ldots, v_n\}$ is the set of thought vertices, each encoding an individual cognitive unit.

- $E = \{e_1, \ldots, e_m\}$ is the set of hyperedges, with $e_j \subseteq V$ able to link an arbitrary non-empty subset of vertices and thereby capture higher-order relations.

- $\Phi : V \rightarrow \mathcal{V}$ assigns each vertex a semantic embedding in space $\mathcal{V}$.

- $\Psi : E \rightarrow \mathcal{E}$ maps each hyperedge to an embedding in space $\mathcal{E}$ that reflects the semantics of the underlying relation.

Thought structure. Each vertex v_i is a quadruple $v_i = (c_i, \tau_i, \sigma_i, \mathbf{x}_i)$ consisting of *content c_i*, type τ_i (premise, hypothesis, *etc.*), *confidence* $\sigma_i \in [0, 1]$ and embedding $\mathbf{x}_i = \mathcal{L}(c_i)$ obtained from a language model encoder $\mathcal{L}$. The confidence score is computed

as:

$$\sigma_i = f_{\text{conf}}(v_i; C, H_t) = \alpha \operatorname{rel}(v_i, C) + \beta \operatorname{cons}(v_i, H_t) + \gamma \operatorname{spec}(v_i), \quad (1)$$

where $\alpha + \beta + \gamma = 1$ and H_t is the hypergraph state at time t.

Hyperedge semantics. Each hyperedge e_j is annotated as $e_j = (S_j, T_j, r_j, w_j, t_j)$ where S_j and T_j are source and target vertex subsets respectively. r_j is a relation label (e.g. *causation, refinement*), $w_j \in \mathbb{R}$ is a weight, and t_j is a temporal stamp. Directionality is indicated when $S_j \neq T_j$. The hyperedge weight is computed as:

$$w_j = \frac{1}{|S_j| \cdot |T_j|} \sum_{v_i \in S_j} \sum_{v_k \in T_j} \operatorname{sim}(\mathbf{x}_i, \mathbf{x}_k) \cdot \operatorname{relevance}(r_j, \tau_i, \tau_k) \quad (2)$$

where $\operatorname{sim}(\mathbf{x}_i, \mathbf{x}_k) = \frac{\mathbf{x}_i \cdot \mathbf{x}_k}{\|\mathbf{x}_i\| \cdot \|\mathbf{x}_k\|}$ is cosine similarity.

Core reasoning operations. HGoT exposes five core operators that together support construction and analysis:

1. **Generate:** Creates a new vertex from context C and k prior thoughts ($C \times V^k \rightarrow V$).

2. **Connect:** Inserts a new hyperedge linking any subset of vertices with a relation label and weight ($2^V \times \mathcal{R} \times \mathbb{R} \rightarrow E$).

3. **Evaluate:** Updates confidence scores ($V \times C \rightarrow [0, 1]$).

4. **Refine:** Outputs an improved version of a thought ($V \times C \rightarrow V$).

5. **Traverse:** Selects the next vertex sequence under strategy $\mathcal{S}$ which can be confidence-guided or relation-guided ($V \times E \times \mathcal{S} \rightarrow V^*$). The traversal strategies are defined as: *confidence-guided* traversal selects $\pi_{\text{conf}}(v_i) = \arg\max_{v_j \in \mathcal{N}(v_i)} \sigma_j$ where $\mathcal{N}(v_i) = \{v_j \in V : \exists e_k \in E \text{ such that } \{v_i, v_j\} \subseteq e_k\}$, *relation-guided* traversal uses $\pi_{\text{rel}}(v_i, r) = \{v_j \in V : \exists e_k \in E \text{ with } r_k = r \text{ and } \{v_i, v_j\} \subseteq e_k\}$; and *multi-objective* traversal applies $\pi_{\text{multi}}(v_i) = \arg\max_{v_j \in \mathcal{N}(v_i)} [\alpha \cdot \sigma_j + \beta \cdot \operatorname{novelty}(v_j) + \gamma \cdot \operatorname{relevance}(v_j, C)]$ where $\operatorname{novelty}(v_j) = 1 - \max_{v_k \in V \setminus \{v_j\}} \operatorname{sim}(\mathbf{x}_j, \mathbf{x}_k)$ and $\operatorname{relevance}(v_j, C) = \operatorname{sim}(\mathbf{x}_j, \mathcal{L}(C))$.

The combination of these elements gives the hypergraph with constructive, analytic, and exploratory capabilities that generalize prominent predecessors: *Chain of Thought* [66], *Tree of Thoughts* [68], and *Graph of Thoughts* [33] are all recovered as special cases when $|e_j| = 2$ with path, tree, or simple-graph topologies, respectively.
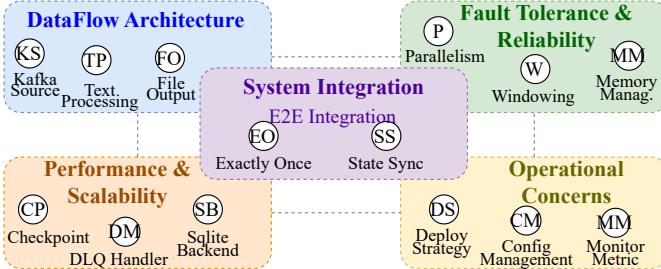
Figure 5: HGoT applied to Wordcount pipeline design. The diagram illustrates how HGoT simultaneously considers multiple interdependent design dimensions through hyperedges (dashed ellipses): Data Flow Architecture (blue), Performance and Scalability (orange), Reliability and Fault Tolerance (green), and Operational Concerns (yellow). Individual thoughts (circles) within each hyperedge represent specific requirements and constraints. The central System Integration hyperedge (purple) captures cross-cutting concerns, such as exactly-once semantics and state synchronization.

Expressivity and complexity. By advancing reasoning from binary relationships to hyperrelationships, HGoT demonstrates a significantly enhanced expressive power compared to CoT, ToT, and GoT. Importantly, its space complexity remains at $O(|V| + \sum_{e_j \in E} |e_j|)$, which is often more efficient than the $O(|V|^2)$ required for enumerating all pairwise relations.

HGOT construction. The process begins by constructing an initial graph with system and user nodes (Algorithm 2, line 11), establishing the fundamental context for reasoning. When RAG is enabled, document nodes containing relevant domain knowledge are incorporated into the graph (Algorithm 2, lines 12-15), creating connections between user requirements and system capabilities. Analysis and plan nodes are then added (Algorithm 2, lines 16-17), representing the system's evolving understanding of the problem and approach to solving it. The resulting structure forms a hypergraph where edges can connect multiple nodes simultaneously, representing complex relationships between different reasoning elements. As visualized in the Figure 3, node types include System (S) containing LLM, User (U) representing user queries, (retrieved domain knowledge (via RAG (R)), Analysis (A) containing query interpretation, Plan (P) holding execution strategies, Design (D) for architectural decisions, and Execution (E) tracking implementation progress. These nodes are interconnected through standard edges and hyperedges, creating a rich structure that captures the multifaceted reasoning necessary for pipeline generation. Algorithm 3 begins by creating *system* (S) and *user* (U) vertices, seeding V_0 with the dialogue context. If RAG is active, document vertices R containing domain knowledge are added and connected to S, U via **Connect**. Subsequent calls to **Generate** and **Refine** populate *analysis* (A) and *plan* (P) vertices, progressively elaborating the solution space. The hyperedge construction process, as outlined in Algorithm 4, employs similarity-based clustering to identify coherent thought groups and automatically determines appropriate relation types and weights. These hyperedges naturally encode constraints such as *"the window size must align with checkpointing intervals"* or *"error-handling strategy depends jointly on sink semantics and broker QoS"*. During traversal, Algorithm 3 (lines 12–

16) alternates a *confidence-guided* and *relation-guided* policy to balance exploitation of promising subgraphs against exploration for novel but relevant alternatives.

Algorithm 3: HGoT Construction

Input: User requirements U, System constraints S, Knowledge base R
Output: Optimal pipeline design D^*

1   $V_0 \leftarrow S \cup U \cup R$
2   $E_0 \leftarrow \text{Conn.}(S \cup U, \text{"context"}) \cup \text{Conn.}(R, S \cup U, \text{"knowledge"})$
3   $H_0 \leftarrow (V_0, E_0, \Phi, \Psi)$
4   **for** $t \leftarrow 1$ **to** *max_iterations* **do**
5     $A_t \leftarrow \{\text{Generate}(\text{context}, V_{t-1}) \text{ for analysis aspects}\}$
6     $V_t \leftarrow V_{t-1} \cup A_t$
7     $P_t \leftarrow \{\text{Generate}(\text{context} \cup A_t, V_t) \text{ for plan components}\}$
8     $V_t \leftarrow V_t \cup P_t$
9     $E_t \leftarrow E_{t-1} \cup \text{Algorithm 4}(V_t)$
10   **if** $t \bmod 2 = 1$ **then**
11   next_vertices $\leftarrow \pi_{\text{conf}}(\text{current\_vertex})$
12   **else**
13   next_vertices $\leftarrow \pi_{\text{rel}}(\text{current\_vertex}, \text{"dependency"})$
14   **foreach** $v \in$ *next_vertices* **do**
15      $v_{\text{refined}} \leftarrow \text{Refine}(v, \text{context})$
16      $V_t \leftarrow V_t \cup \{v_{\text{refined}}\}$
17   **if** *convergence_criterion_met*(H_t) **then**
18   **break**
19   $D^* \leftarrow \text{extract\_optimal\_design}(H_t)$
20   **return** D^*

Algorithm 4: Hyperedge Construction

1   Compute pairwise similarities $S_{i,j} = \text{sim}(v_i, v_j)$ for all $v_i, v_j \in V$;
2   Apply hierarchical clustering to obtain candidate groups G;
3   **for** *each cluster* $g_k \in G$ *with* $|g_k| \geq 2$ **do**
4   Identify the common theme θ_k across thoughts in g_k;
5   Assign relation type r_k based on θ_k;
6   Compute weight $w_k = \frac{1}{|g_k|} \sum_{v_i, v_j \in g_k} S_{i,j}$;
7   Create hyperedge $e_k = (g_k, r_k, w_k, 0, t)$;

Example. Figure 5 illustrates the hypergraph decomposition of the Flink pipeline into six critical hyperedges that capture the multi-dimensional relationships between architectural components:

$e_1 = (\{KS, TP, FO\}, \emptyset, \text{"data\_flow"}, w_1, 0, t_1)$

$e_2 = (\{P, W, MM\}, \emptyset, \text{"performance\_optimization"}, w_2, 0, t_2)$

$e_3 = (\{CP, DM, SB\}, \emptyset, \text{"fault\_tolerance"}, w_3, 0, t_3)$

$e_4 = (\{MM, CM, DS\}, \emptyset, \text{"operational\_Concern"}, w_4, 0, t_4)$

$e_5 = (\{P, CP, W\}, \{EO\}, \text{"perf\_reliability\_tradeoff"}, w_5, 1, t_5)$

$e_6 = (\{KS, CP, SB, DM, EO, SS\}, \{EC\}, \text{"system\_integration"}, w_6, 1, t_6)$

Hyperedge e_1 represents the core data flow architecture, connecting the Kafka Source (KS) through Text Processing (TP) to File Output (FO), forming the primary ingestion and transformation path. Hyperedge e_2 captures performance and scalability constraints, where the parallelism configuration P_{4-8-2} must be harmonized with the 30-second windowing strategy W_{30s} and task memory allocation (MM) to avoid bottlenecks. Hyperedge e_3 models the fault-tolerance mechanism, requiring tight coordination between 10-second checkpoint intervals CP_{10s}, Dead Letter Queue (DLQ) handling, and State Backend (SB) persistence to ensure recovery consistency.

The directed hyperedge e_5 is critical: it represents the performance–reliability trade-off, where the interplay between parallelism levels, checkpoint frequency, and window duration col-

Algorithm 5: Execute Step With Retry

Input : executor, step, plan
Output: Result of the executed step

1 **Function** ESR(*executor, step, plan*):
2   retries, maxRetries $\leftarrow$ 0, 5;
3   **while** *retries < maxRetries* **do**
4   **try:**
5   result $\leftarrow$ ExecuteStep(step.action, plan.query,
            step.dependencies);
6   **return** *result*;
7   **catch:**
8   APIError e
9   **if** *e.isRateLimit* **then**
10   delay $\leftarrow$ baseDelay $\times 2^{retries} \times (0.5 + random())$;
11   Sleep(delay);
12   **else if** *e.isQuotaExceeded* **then**
13   SwitchToNextModel(executor);
14   retries $\leftarrow$ retries + 1;
15   **return** GenerateFallbackResult(*step*);
16 **Function** SwitchToNextModel(*executor*):
17   currentIndex $\leftarrow$ executor.currentModelIndex;
18   modelPool $\leftarrow$ executor.modelPool;
19   **if** *currentIndex < |modelPool| − 1* **then**
20   executor.currentModelIndex $\leftarrow$ currentIndex + 1;
21   executor.activeModel $\leftarrow$ modelPool[currentIndex + 1];
22   **else**
23   executor.currentModelIndex $\leftarrow$ 0;
24   executor.activeModel $\leftarrow$ modelPool[0];

---

lectively determines the feasibility of exactly-once (EO) processing semantics. Violation of this constraint leads to state inconsistency or degraded throughput.

Most importantly, hyperedge e_6 highlights the main system integration constraint, marked in purple in Figure 5. It connects six key components: Kafka Source (KS), checkpointing (CP$_{10s}$), state backend (SB), DLQ, exactly-once semantics (EO), and snapshotting (SS), with emergent consistency (EC) as the dependent outcome. This hyperedge ensures end-to-end correctness and is vital for reliable and consistent stream processing across various components.

Since these five decisions exist within a single higher-order constraint, any change, such as reducing the window from 30 seconds to 10 seconds, prompts a review of the other four parameters. This aim is to maintain exactly-once semantics through synchronized adjustments. The checkpoint interval must maintain a 1:1 ratio (30 seconds to 10 seconds), parallelism must remain optimal for the new window, and the DLQ policy needs to adjust timeout thresholds. This synchronized adjustment isn't feasible in pairwise-edge frameworks (CoT, ToT, GoT). It demonstrates how HGoT guides AutoStreamPipe towards a coherent design point D^*. The process culminates in the executable Flink specification presented in the box 1.

Step 2: Agent Pool Configuration. Multiple models from different providers are initialized (Algorithm 2, lines 4-5), including specialized models from OpenAI, Mistral, Anthropic, and Groq, as depicted in Figure 3. These models are organized into a managed pool with sophisticated load balancing capabilities that distribute tasks based on model strengths, availability, and quota consumption. Failover mechanisms are configured to automatically switch between models when rate limits or other failures are encountered, ensuring continuous operation even when individual providers experience limitations. The agent selection system incorporates a static configuration based on known model capabilities and a dynamic adaptation based on observed performance, creating an evolving selection mecha-

nism that optimizes quality and reliability. This agent pool represents a key innovation in AutoStreamPipe, transcending the limitations of single-model approaches by creating a heterogeneous AI team with complementary capabilities.

Step 3: Multi-Agent Coordination Setup. A resilient plan executor is created (Algorithm 2, line 20) that manages the assignment of tasks to specific models based on their capabilities and availability. The coordination system implements error handling mechanisms, as visualized in Figure 3 by red dashed lines connecting to the Error Handler (EH) node. These connections represent the system's ability to detect failures at any point in the process and initiate appropriate recovery actions. Different steps in the pipeline generation process are assigned to other agents based on their specialized capabilities (e.g., using models with strong code generation abilities for implementation steps while employing models with superior planning capabilities for architectural design). This specialization maximizes the quality of outputs while minimizing the likelihood of failures.

The coordination system also incorporates evaluation mechanisms that assess the quality of agent outputs, enabling the system to request refinements when necessary. This phase creates a robust cognitive framework that enables sophisticated reasoning and resilient multi-model collaboration, thereby forming the computational foundation for generating complex pipelines.

4.4. Phase 3: Resilient Execution and Artifact Management

The final phase executes the plan while ensuring resilience and managing artifacts. This phase is implemented across Algorithm 2 (lines 20-35) and Algorithm 5, with the resilient model handler layer in Figure 3 directly corresponding to Algorithm 5.

Step 1: Resilient model handler. The resilient model handler employs four key strategies. Multi-agent model rotation is implemented through the SwitchToNextModel function (line 13). The jitter-induced exponential backoff is defined in lines 9-11, using the formula shown in line 10. The handling of the rate limit and graceful degradation are managed through the handling of conditional errors (lines 9-14) and the generation of the fallback result (line 15).

Step 2: Step-by-Step Plan Execution. The step-by-step plan execution ensures that steps are performed only when their dependencies are satisfied (Algorithm 2, lines 16-18), allowing each phase to build upon the outputs of its prerequisites. Each step, such as analyze_complexity or design, incorporates retry mechanisms to handle transient failures (line 19). The results of each step are saved methodically (line 20), creating a persistent record of the process. When results include Java code, as is common in steps like generate_pipeline, the system extracts the code for separate storage to facilitate deployment (lines 21-23). Each step is marked as completed upon successful execution (line 23), updating the execution plan's state to reflect progress. The visual execution process, shown in Figure 3, illustrates the progression, with arrows indicating dependencies between steps. This structured execution ensures that complex pipeline generation proceeds logically, with each step receiving the necessary inputs from prior operations.

Step 3: Artifact Management. We organize generation outputs into coherent, usable deliverables. The SP pipeline is extracted from relevant steps, particularly `generate_pipeline`, and formatted according to language-specific conventions. The step results are saved in a structured JSON format, providing a complete record for quality evaluation or debugging. The final response is synthesized (Algorithm 2, 24) into a narrative detailing the pipeline architecture, implementation, and operational aspects. A session summary is created (line 25), cataloging all generated artifacts with explanations of their purposes and relationships. Additionally, the memory of the interaction is saved for future reference (line 26), enabling the system to build upon prior experiences when handling similar requests. This process transforms the raw outputs of individual steps into a cohesive collection of interrelated resources that collectively satisfy the user's request.

Finally, the output generation produces deliverables tailored to the user's needs. These include an SP pipeline solution, consisting of executable code, configuration files, supporting resources, and comprehensive documentation that covers architecture, implementation, deployment, and operations. All artifacts are returned as the final output (Algorithm 2, line 27), marking the transformation of a natural language request into a production-ready implementation through multi-agent collaboration and resilient execution.

4.5. Integration and System Flow

The complete AUTOSTREAMPIPE system integrates all three phases into a seamless workflow, transforming natural language queries into deployable SP pipelines. Algorithm 2 serves as the central controller, orchestrating the entire process from initialization to final output generation. It begins with establishing the model infrastructure, including backup models for resilience, then proceeds through query analysis, graph building, plan execution, and artifact management.

The AUTOSTREAMPIPE architecture significantly advances automated SP pipeline generation, bridging the gap between natural language expressions of intent and production-ready implementation. By combining structured reasoning through hypergraphs, multi-agent collaboration across model providers, and resilience mechanisms, the system achieves a level of reliability and capability that exceeds traditional single-model approaches. The detailed algorithms formally define the system's operation, demonstrating how each visual component in Figure 3 is implemented in practice.

5. Performance Evaluation

5.1. Experimental Setup

We comprehensively evaluated the AUTOSTREAMPIPE, testing its planning and resilient execution capabilities. All experiments were conducted on a system equipped with an Intel Core i7-13700K processor (16 cores, 3.4 GHz) and 64GB of RAM, running Ubuntu 22.04 LTS. AUTOSTREAMPIPE was developed in Python (v3.12.7) and integrates LangChain (v0.3.11) for LLM

Word Count Pipeline (Partial Description Version)
Create a Flink streaming application for text processing:

- **Source:** Write to Kafka topic
- **Input format:** Plain text messages
- **Process:** Split messages into words by whitespace and count frequencies
- **Windowing:** Aggregate every 30 seconds
- **Output:** Local file at `word-counts.txt`

Box 2: Word Count Pipeline (Partial Description Version)

support. For testing the generated pipelines, we used the latest stable versions of Apache Flink (v1.20.1), Storm (v2.8.0), and Spark (v3.5.5). We evaluated two types of LLM: Codestral Mamba [73] and llama-3.3 [74] as our Open Source (OS) option, and ChatGPT-4o-mini [75] and Claude-Haiku [76] as our Closed Source (CS) option, with detailed comparisons available in our evaluation table. The AUTOSTREAMPIPE codebase comprises about 4,000 lines of Python for the framework implementation and 15,000 lines of Java for the applications.

5.1.1. Dataset and Queries.

In our paper, we develop a benchmark suite that contains eight diverse SP applications specifically designed for pipeline design evaluation. This suite includes self-defined applications + word count [77, 78, 79] (see Figure 6). For each application, we define two query types: full- and partial-information queries. For example, the word count application is illustrated by the full information query in the box 1, where all relevant information is provided in the prompt, and the partial information in the box 2, where some details are missing. All queries are available in our repository[1]. All benchmarks are available on the AUTOSTREAMPIPE GitHub repository[2]. We carefully selected these applications to design their pipelines, providing a balanced mix of three simple, three medium, and two complex pipelines that target commonly used SPEs, along with a custom data generator for each pipeline. To account for variability in the generation process, we produced each pipeline five times for each of the three SPEs (Flink, Storm, and Spark).

5.1.2. Baseline Systems.

We compared our system against the following alternatives:

---

[1] https://github.com/Anonymous0-0paper/SWG/blob/master/Query_docs.txt

[2] https://github.com/Anonymous0-0paper/SWG

Table 3: LLM Models. MTok: Millions of Tokens, (I/O): Input / Output Token

Type	Provider	LLM Model	API	Price ($ /(I/O)MTok)
OS	Mistral AI	Codestral Mamba	open-codestral-mamba	-
	Meta	llama-3.3	llama-3.3-70b-versatile	-
CS	Anthropic	Claude	claude-3-5-haiku-20241022	$0.80 / $4.00
	Open AI	gpt-4o-mini	gpt-4o-mini-2024-07-18	$0.15 / $0.60

(a) Log aggregator application (simple)

(b) CSV data transformation (simple)

(c) Industrial equipment predictive maintenance (medium)

(d) Temperature monitoring [70] (medium)

(e) Event filtering (medium)

(f) Image compression [71] (complex)
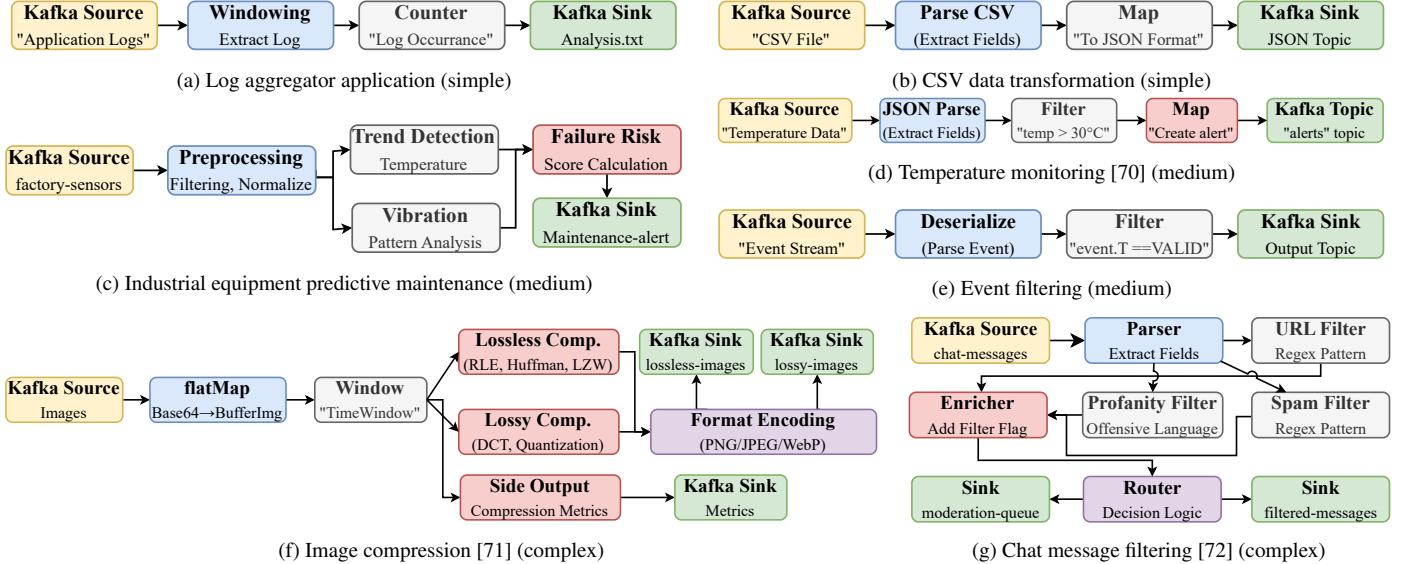
(g) Chat message filtering [72] (complex)

Figure 6: Overview of various data processing pipelines used in different applications, showcasing their architecture

- **Base-LLM**: Direct queries to the LLM without HGoT, planning, and resilience features.

- **CoT Planning**: Our system with query analysis and CoT planning without resilience mechanisms.

- **GoT based**: Standard GoT implementation without hypergraph capabilities and our query analyzer.

- **AutoStreamPipe (ASP)**: Complete implementation with HGoT, query analysis, planning, and resilient execution.

5.2. Evaluation Metrics

We evaluated our system using several key metrics. In some cases, we relied on unit tests because the advanced capabilities of LLM often exceed the scope of standard automated evaluation metrics for general natural language tasks [80, 81].

Processing Time. Total time required to process a query, measured in seconds. **Response Completeness**. Percentage of query requirements addressed in the response.

Error-free Score (EFS). To comprehensively assess the quality of a generated pipeline, we introduce a metric called the Error-Free Score (EFS). This metric evaluates the pipeline's accuracy (meaning how often the model generates correct pipelines) and quality by considering multiple errors that can occur during its creation and execution. The EFS is designed to provide a

more nuanced and fine-grained assessment of pipeline quality, surpassing simple measures such as compilation success rates. The EFS is calculated using the following equation:

$$\text{EFS} = \frac{1}{3}\left(\frac{1}{1+S} + \frac{1}{1+L} + \frac{1}{1+R}\right) \quad (3)$$

where S is the number of syntax errors in the generated pipeline, L means the number of logical errors (correct syntax but incorrect algorithm/logic), and R is the number of runtime errors when executing the code (e.g., null pointer dereferencing). The EFS produces a score between 0 and 1, where 1 represents an entirely error-free pipeline. This metric provides a fine-grained assessment of pipeline quality beyond simple compilation success rates.

5.3. Evaluation Results

Error-Free Score. Table 4 compares EFS in three pipeline complexities (Simple, Medium, Complex) and three SPEs (Flink, Storm, Spark). Each row displays the EFS for four approaches: Base-LLM, CoT Planning, GoT-Based, and AutoStreamPipe, with an average value at the bottom of each complexity group.

AutoStreamPipe reaches an average EFS of 0.98 for simple pipelines, indicating near-perfect performance. This high score is due to the pipeline's simplicity, which minimizes the occurrence of syntax, logic, and runtime errors. In contrast, CoT Planning achieves an EFS of 0.65 and Base-LLM 0.46, highlighting the limitations of these methods in handling even simple tasks without error. The significant improvement in AutoStreamPipe is primarily due to the HGoT and advanced query analysis, which enhance the accuracy and consistency of the generated pipelines. In medium pipelines, AutoStreamPipe maintains a high average EFS (0.73), significantly outperforming GoT Based (0.54), CoT Planning (0.44), and Base-LLM (0.36). This performance difference arises because medium-complexity pipelines introduce more intricate logical relationships and data transformations, which traditional methods struggle to capture

Table 4: Error-free Score (EFS) comparison. PL: Pipeline

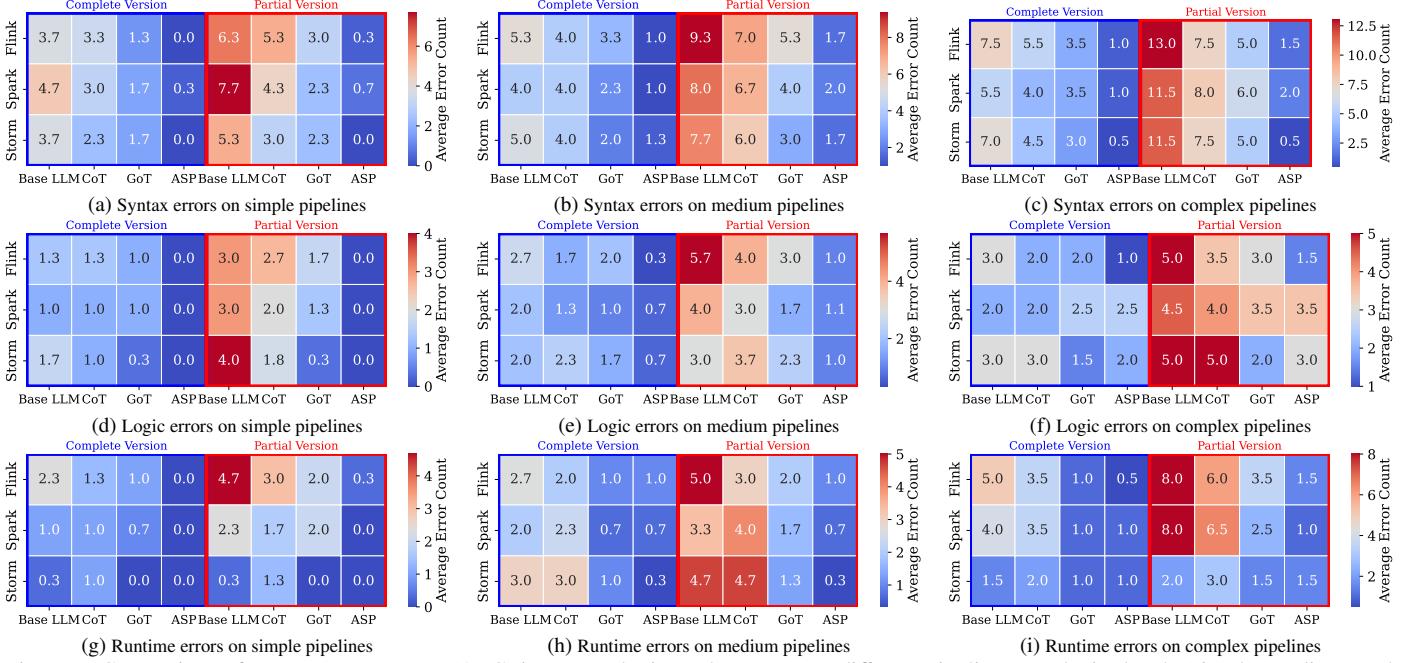| PL Type	SPE	Base-LLM	CoT Planning	GoT Based	AutoStreamPipe
Simple	Flink	0.36	0.41	0.51	**1**
	Storm	0.54	0.56	0.78	**1**
	Spark	0.47	0.51	0.65	**0.94**
Average		0.46	0.49	0.65	**0.98**
Medium	Flink	0.26	0.32	0.4	**0.68**
	Storm	0.27	0.28	0.5	**0.73**
	Spark	0.32	0.36	0.52	**0.69**
Average		0.28	0.32	0.47	**0.7**
Complex	Flink	0.18	0.24	0.36	**0.63**
	Storm	0.27	0.27	0.44	**0.6**
	Spark	0.24	0.26	0.39	**0.54**
Average		0.23	0.26	0.4	**0.59**

Figure 7: Comparison of average error counts (AEC) in syntax, logic, and error across different pipeline complexity levels (simple, medium, and complex) on complex and partial information versions for four distinct approaches (Base LLM, CoT, GoT, and AUTOSTREAMPIPE (ASP)). These plots illustrate how error frequencies vary with increasing pipeline complexity and approach, providing insights into performance differences.

accurately. The HGoT's multi-way reasoning and robust execution planning in AUTOSTREAMPIPE enable it to handle these challenges more effectively. Performance declines for all approaches as pipeline complexity increases. For complex pipelines, AUTOSTREAMPIPE achieves an average EFS of 0.59. Although this score is lower than for simpler pipelines, it remains substantially higher than CoT Planning (0.42) and Base-LLM (0.23–0.30 range). The reduced performance is expected due to the inherent challenges of managing complex interdependencies. However, the structured reasoning of the HGoT and the resilient multi-agent execution in AUTOSTREAMPIPE ensure that it still achieves a 50% improvement over CoT Planning and a 64% improvement over Base-LLM. Overall, AUTOSTREAMPIPE delivers the best results in every scenario due to HGOT and our query analyzer. In the best case (Simple pipelines), it achieves an average EFS of 0.98, and even in the worst case (Complex pipelines), it reaches 0.59, a 50% improvement over CoT Planning and a 64% improvement over the Base-LLM.

Error Distribution and Composition. Figures 7 and 8 present a detailed analysis of error patterns across various pipeline generation approaches and SPEs, to show how well each method handles different types of errors. Figure 7 visualizes the average counts of syntax, logic, and runtime errors across three levels of pipeline complexity (simple, medium, complex) and two input settings: complete and partial information. Across all configurations, we observe a consistent trend: as we move from the Base LLM to the proposed AUTOSTREAMPIPE (ASP), the average error counts decrease substantially. This improvement is especially evident under the partial information setting, where approaches must infer or complete missing pipeline specifications. Syntax errors are the most dominant across all models,

particularly in Base LLM and CoT, which lack structured planning mechanisms. These errors are mitigated more effectively by ASP due to its robust multi-agent coordination and planning capabilities. In contrast, logic errors show a moderate reduction as we move from simpler to more advanced methods. ASP's enhanced context understanding and error-aware planning significantly reduce these issues, particularly in medium-to complex-sized pipelines.

Runtime errors, while fewer in number, represent critical failures in executable pipelines. These errors are persistently challenging for all methods, but ASP demonstrates a clear advantage, particularly in complex pipelines with partial information, where it maintains the lowest runtime error counts. Importantly, the gap between the complete and partial versions increases with the complexity of the pipeline. All baseline models degrade significantly under partial inputs, whereas ASP maintains low error rates and high stability, demonstrating its robustness in uncertain and ambiguous input scenarios.

Figures 7a-7i complement this analysis by presenting the average error composition for three SPEs. Here, Syntax errors dominate the total error count in Base-LLM because this method lacks structured reasoning and robust error handling. Logic and runtime errors are less frequent but significant, indicating that even basic pipeline generation methods can occasionally produce syntactically correct but logically flawed code. As the approaches become more advanced, the AUTOSTREAMPIPE method achieves the lowest overall error counts, with balanced reductions across all error types. The AUTOSTREAMPIPE minimizes syntax errors through improved planning and addresses logical inconsistencies through enhanced reasoning, as well as runtime issues via resilient execution strategies. Figures 7a-7i,
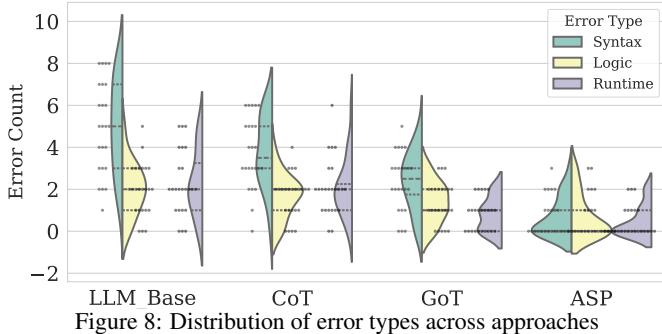
12

Figure 8: Distribution of error types across approaches

Figure 10: Comparison of quality of pipelines across four distinct approaches for stateless and stateful pipelines.

and 8 underscore the effectiveness of the AUTOSTREAMPIPE in minimizing errors and highlight the importance of selecting an appropriate SPE to optimize pipeline quality.

Error Reduction by Difficulty Level. Figure 9 illustrates the reduction in average errors across three pipeline difficulty levels. The AUTOSTREAMPIPE consistently achieves the lowest error rates across all difficulty levels, demonstrating its superior performance compared to the other methods. This consistency is due to the AUTOSTREAMPIPE's ability to adapt its reasoning and execution strategies according to pipeline complexity, unlike other methods that struggle as pipelines become more intricate. As pipeline complexity increases, the average number of errors rises for all approaches, but the gap between Base LLM and AUTOSTREAMPIPE widens significantly, particularly for complex pipelines. This pattern highlights that basic methods are prone to generating errors when faced with complex data transformations and dependencies. AUTOSTREAMPIPE's multi-agent coordination and hypergraph reasoning help maintain lower error rates. Intermediate approaches, such as CoT and GoT, demonstrate gradual improvements over the baseline, highlighting their incremental effectiveness. Furthermore, the consistent reduction in errors from Base LLM to AUTOSTREAMPIPE underscores the cumulative benefits of integrating advanced strategies into pipeline generation. Overall, Figure 9 highlights the critical role of pipeline complexity in influencing error rates and demonstrates the robustness of AUTOSTREAMPIPE in achieving reliable performance across varying levels of complexity. These insights are crucial for understanding the practical applications of pipeline generation methods and selecting the most suitable strategy for various complexities.

Stateless and Stateful Pipeline Quality. The pipeline quality generated using AUTOSTREAMPIPE was evaluated using automated compilation tests. Lower total errors mean higher quality. Figure 10 analyzes the quality of pipelines generated by
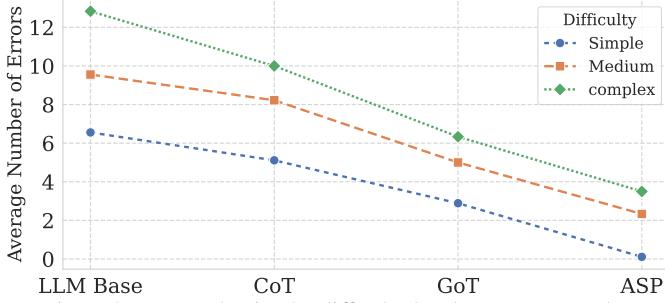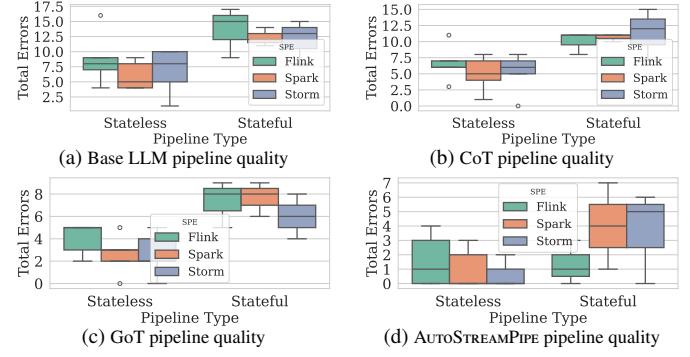
four approaches: Base LLM, CoT, GoT, and AUTOSTREAMPIPE across two pipeline types (Stateless and Stateful) and three SPEs. The figure is divided into four subplots (10a) - (10d), each representing the error distributions for a specific approach. Across all approaches, stateless pipelines consistently exhibit lower error rates than stateful pipelines, reflecting the added complexity of managing stateful operations. The Base LLM approach in Figure (10a) exhibits the highest error rates, with significant variability and pronounced outliers, particularly in stateful pipelines. The CoT approach in Figure (10b) demonstrates noticeable improvements, reducing median errors and variability. Meanwhile, GoT in Figure (10c) further refines performance, especially for stateful pipelines. The AUTOSTREAMPIPE in Figure (10d) achieves the best results, with the lowest median error counts, a tight interquartile range, and minimal outliers, indicating consistent and reliable performance. These findings demonstrate the effectiveness of AUTOSTREAMPIPE in mitigating errors, particularly in complex stateful scenarios, and highlight the importance of selecting an appropriate SPE to optimize pipeline quality.

Iteration and Improvement Relation. Figure 11 presents two complementary views of improvement outcomes in all SPEs. In Figure 11a, we compare the proportion of fully fixed solutions (upper segment of each bar) against partially fixed solutions (lower segment), which gives insight into how often each SPE achieves complete resolution of the issue. In Figure 11b, we illustrate the number of iterations (x-axis) required to achieve a particular percentage of improvement (y-axis), with each point representing a single experimental run. The bubble size encodes the magnitude of the error reduction (i.e., the number of errors addressed), while distinct colors identify the SPE, and marker shapes indicate whether the final solution was fully fixed. Taken together, we understand from figures 11a and 11b that elucidate which SPEs tend to yield complete fixes more consistently, that 3 or 4 iterations are typically necessary to reach high levels of improvement, and the relative difficulty of resolving errors across different SPE scenarios.

Development Time: Speeding Up Production. The results show that AUTOSTREAMPIPE significantly reduces development effort. For simple pipelines, its development time (10 to 20 minutes) is similar to visual tools like NiFi. However, as task complexity increases, the advantage becomes much clearer. For

Figure 9: Error reduction by difficulty level across approaches

(a) Distribution of fully fixed cases by SPE

(b) Iterations vs Improvement (Bubble size: Error reduction)
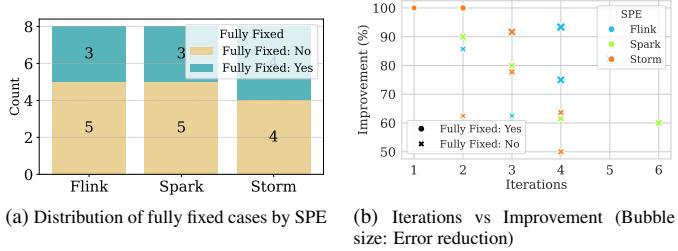
Figure 11: Overview of fully fixed outcomes and iterative improvements across different SPEs.

Table 5: Development Time and Performance Metrics by Pipeline Complexity

Complexity	Development Time (min)			Throughput (k events/s)			Latency P99 (ms)		
	ASP	NiFi	Manual	ASP	NiFi	Manual	ASP	NiFi	Manual
Simple	10–20	10–20	40–90	150	138	153	12.8	18.3	10.8
Medium	15–30	20–40	70–140	98	81	105	28.1	40.5	25.4
Complex	25–45	45–70	150–300	72	60	78	55.7	84.7	54.4

Note: Throughput and latency metrics were measured after any errors in the generated pipelines.

complex pipelines, AUTOSTREAMPIPE takes only 25 to 45 minutes, making it up to 1.6× faster than NiFi (which takes 45 to 70 minutes). This improvement is especially noticeable compared to manual coding. An expert developer might spend up to five hours (300 minutes) creating a complex pipeline, while AUTOSTREAMPIPE completes the same task in about 35 minutes, achieving a remarkable 6.3× reduction in development time. This speedup is due to the framework's ability to convert high-level user intent into efficient, platform-specific code, effectively addressing a key challenge mentioned in Section 1.

While speeding up development is a key goal, the generated pipelines must also perform well in production environments. Our analysis shows that AUTOSTREAMPIPE achieves a strong balance between automation and runtime efficiency. When compared to the hand-written baseline, the pipelines produced by AUTOSTREAMPIPE have only a slight performance overhead. The throughput is, on average, 2 to 8 percent lower than that of the manually optimized code. Similarly, the end-to-end latency increases by 2 to 18 percent. This demonstrates that the automated generation process incurs minimal performance cost, confirming its effectiveness in challenging, real-world situations. In contrast to the visual, low-code approach of Apache NiFi, AUTOSTREAMPIPE shows significant performance improvements. Across all levels of complexity, our generated pipelines achieve 15 to 20 percent higher throughput and 30 to 35 percent lower latency than those created with NiFi. This indicates that AUTOSTREAMPIPE not only streamlines development but also produces pipelines that are much more efficient than those from traditional low-code tools.

In summary, AUTOSTREAMPIPE delivers the fast development experience of a low-code platform while ensuring that pipelines maintain nearly optimal performance, comparable to that of expert manual implementation.

6. Conclusion

In this work, we introduce AUTOSTREAMPIPE, a novel framework for automating the generation of data stream processing pipelines employing modern LLMs. AUTOSTREAMPIPE directly addresses the challenges faced by domain experts who lack programming expertise but require sophisticated SP pipelines to meet their needs. Our approach is built on three key innovations: **(1)** a specialized user query analyzer that performs intent detection and parameter extraction to construct tailored execution plans, **(2)** a novel cognitive reasoning framework, which we term the *Hypergraph of Thoughts*, that decomposes high-level semantic queries into executable subtasks and synthesizes pipelines across diverse stream processing engines, and **(3)** a fault-tolerant multi-agent execution layer that coordinates specialized LLM agents to collaboratively produce and refine pipelines with error recovery mechanisms. Our evaluation of eight diverse workloads and three SPEs demonstrates that AUTOSTREAMPIPE reduces error rates by 5.19× and lowers pipeline development time by 6.3× and reduces the response time on average 27% compared to state-of-the-art baselines. The system consistently outperforms baselines in terms of correctness (executable rate) and stability (output consistency). We also propose the Error-Free Score (EFS) metric to assess the proportion of fully correct and executable pipelines, which offers a more comprehensive evaluation criterion than syntactic validation alone. Furthermore, the integration of persistent memory and adaptive agent orchestration enables the system to maintain contextual fidelity across complex, multi-turn interactions. Future work will include: **(i)** support for real-time schema evolution and dynamic workloads, and **(ii)** integration with self-healing mechanisms for runtime fault adaptation.

References

[1] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R. H. Campbell, Samza: stateful scalable stream processing at linkedin, Proceedings of the VLDB Endowment 10 (12) (2017) 1634–1645.

[2] V. Cardellini, F. Lo Presti, M. Nardelli, G. R. Russo, Runtime adaptation of data stream processing systems: The state of the art, ACM Computing Surveys 54 (11s) (2022) 1–36.

[3] L. Yang, A. Shami, A multi-stage automated online network data stream analytics framework for iiot systems, IEEE Transactions on Industrial Informatics 19 (2) (2023) 2107–2116.

[4] A. Younesi, E. Oustad, M. Abolnejadian, M. Ansari, A. Ejlali, MoTiCPS: Energy Optimization on Multi-Objective Task Scheduling in IoT-Integrated Cyber-Physical Systems , IEEE Transactions on Sustainable Computing (01) (2025) 1–12.

[5] R. A. L. Torres, M. Ladeira, A proposal for online analysis and identification of fraudulent financial transactions, in: 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), 2020, pp. 240–245.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al., The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proceedings of the VLDB Endowment 8 (12) (2015) 1792–1803.

[7] N.-R. Contributors, Node-red, accessed: 2025-02-13 (2013).
URL https://github.com/node-red

[8] Apache Software Foundation, Apache nifi, https://github.com/apache/nifi (2025).

[9] P. Tucker, K. Tufte, V. Papadimos, D. Maier, Nexmark–a benchmark for queries over data streams (draft), Technical report (2008).

[10] A. M. Garcia, D. Griebler, C. Schepke, L. G. Fernandes, Spbench: a framework for creating benchmarks of stream processing applications, Computing 105 (5) (2023) 1077–1099.

[11] A. Shukla, S. Chaturvedi, Y. Simmhan, Riotbench: An iot benchmark for distributed stream processing systems, Concurrency and Computation: Practice and Experience 29 (21) (2017) e4257.

[12] M. V. Bordin, D. Griebler, G. Mencagli, C. F. Geyer, L. G. L. Fernandes, Dspbench: A suite of benchmark applications for distributed data stream processing systems, IEEE Access 8 (2020) 222900–222917.

[13] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, H. Plattner, Espbench: The enterprise stream processing benchmark, in: Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2021, pp. 201–212.

[14] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang, The hibench benchmark suite: Characterization of the mapreduce-based data analysis, in: 2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010), IEEE, 2010, pp. 41–51.

[15] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear road: a stream data management benchmark, in: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, 2004, pp. 480–491.

[16] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al., Benchmarking streaming computation engines: Storm, flink and spark streaming, in: 2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW), IEEE, 2016, pp. 1789–1792.

[17] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark, in: Proceedings of the 12th ACM international conference on computing frontiers, 2015, pp. 1–8.

[18] R. Lu, G. Wu, B. Xie, J. Hu, Stream bench: Towards benchmarking modern distributed stream computing frameworks, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, IEEE, 2014, pp. 69–78.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al., Bigdatabench: A big data benchmark suite from internet services, in: 2014 IEEE 20th international symposium on high performance computer architecture (HPCA), IEEE, 2014, pp. 488–499.

[20] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, H.-A. Jacobsen, Bigbench: towards an industry standard benchmark for big data analytics, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, p. 1197–1208.

[21] T. G. Armstrong, V. Ponnekanti, D. Borthakur, M. Callaghan, Linkbench: a database benchmark based on the facebook social graph, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, p. 1185–1196.

[22] P. Agnihotri, B. Koldehofe, R. Heinrich, C. Binnig, M. Luthra, Pdsp-bench: A benchmarking system for parallel and distributed stream processing (2025). arXiv:2504.10704.
URL https://arxiv.org/abs/2504.10704

[23] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., Gpt-4 technical report, arXiv preprint arXiv:2303.08774 (2023).

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al., Llama: Open and efficient foundation language models, arXiv preprint arXiv:2302.13971 (2023).

[25] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, et al., Code llama: Open foundation models for code, arXiv preprint arXiv:2308.12950 (2023).

[26] Apache Software Foundation, Apache flink, https://github.com/apache/flink (2025).

[27] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink: Stream and batch processing in a single engine, The Bulletin of the Technical Committee on Data Engineering 38 (4) (2015).

[28] Apache Software Foundation, Apache storm, `https://github.com/apache/storm` (2025).

[29] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., Storm@ twitter, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, 2014, pp. 147–156.

[30] Apache Software Foundation, Apache kafka, `https://github.com/apache/kafka` (2025).

[31] Apache Software Foundation, Apache spark, `https://github.com/apache/spark` (2025).

[32] C. Qu, S. Dai, X. Wei, H. Cai, S. Wang, D. Yin, J. Xu, J.-R. Wen, Tool learning with large language models: A survey, Frontiers of Computer Science 19 (8) (2025) 198343.

[33] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, M. Podstawski, L. Gianinazzi, J. Gajda, T. Lehmann, H. Niewiadomski, P. Nyczyk, et al., Graph of thoughts: Solving elaborate problems with large language models, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 38, 2024, pp. 17682–17690.

[34] M. Franklin, E. Tyson, J. Buckley, P. Crowley, J. Maschmeyer, Auto-pipe and the x language: a pipeline design tool and description language, in: Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, 2006, pp. 10 pp.–. `doi:10.1109/IPDPS.2006.1639353`.

[35] C. Yang, Y. Zheng, X. Tu, R. Ala-Laurinaho, J. Autiosalo, O. Seppänen, K. Tammi, Ontology-based knowledge representation of industrial production workflow, Advanced Engineering Informatics 58 (2023) 102185.

[36] T. Coleman, H. Casanova, R. F. da Silva, Wfchef: Automated generation of accurate scientific workflow generators, in: 2021 IEEE 17th International Conference on eScience (eScience), IEEE, 2021, pp. 159–168.

[37] T. Coleman, H. Casanova, R. F. Da Silva, Automated generation of scientific workflow generators with wfchef, Future Generation Computer Systems 147 (2023) 16–29.

[38] I.-C. Donca, O. P. Stan, M. Misaros, D. Gota, L. Miclea, Method for continuous integration and deployment using a pipeline generator for agile software projects, Sensors 22 (12) (2022) 4637.

[39] B. Haynes, A. Cheung, M. Balazinska, Pipegen: Data pipe generator for hybrid analytics, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, 2016, pp. 470–483.

[40] W. Lugmayr, V. Kotov, N. Goessweiner-Mohr, J. Wald, F. DiMaio, T. C. Marlovits, Starmap: a user-friendly workflow for rosetta-driven molecular structure refinement, Nature protocols 18 (1) (2023) 239–264.

[41] C. Carthen, A. Zaremehrjardi, V. Le, C. Cardillo, S. Strachan, A. Tavakkoli, F. C. Harris, S. M. Dascalu, Orchestrating apache nifi/minifi within a spatial data pipeline, in: 2023 IEEE/ACIS 21st International Conference on Software Engineering Research, Management and Applications (SERA), 2023, pp. 366–371.

[42] K. Wnęk, P. Boryło, A data processing and distribution system based on apache nifi, Photonics 10 (2) (2023).

[43] J. Liu, E. Braun, C. Düpmeier, P. Kuckertz, D. S. Ryberg, M. Robinius, D. Stolten, V. Hagenmeyer, A generic and highly scalable framework for the automation and execution of scientific data processing and simulation workflows, in: 2018 IEEE International Conference on Software Architecture (ICSA), IEEE, 2018, pp. 145–14510.

[44] StreamSets, Streamsets on github, accessed: February 10, 2025 (2025).
URL `https://github.com/streamsets`

[45] O. Foundation, Node-red, accessed: February 10, 2025] (2025).
URL `https://nodered.org/`

[46] A. Brito, A. Martin, C. Fetzer, I. Rocha, T. Nóbrega, Streammine3g oneclick – deploy and monitor esp applications with a single click, in: 2013 42nd International Conference on Parallel Processing, 2013, pp. 1014–1019.

[47] C. S. Ranganathan, R. Raman, V. K. Pandey, S. Kavitha, M. Muthulekshmi, K. Gopalakrishnan, Ingestion of google cloud platform data using dataflow, in: 2023 7th International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC), IEEE, 2023, pp. 338–342.

[48] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, G. Li, A survey on code generation with llm-based agents, arXiv preprint arXiv:2508.00083 (2025).

[49] J. Wang, Y. Chen, A review on code generation with llms: Application and evaluation, in: 2023 IEEE International Conference on Medical Artificial Intelligence (MedAI), IEEE, 2023, pp. 284–289.

[50] M. Nejjar, L. Zacharias, F. Stiehle, I. Weber, Llms for science: Usage for code generation and data analysis, Journal of Software: Evolution and Process 37 (1) (2025) e2723.

[51] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., Starcoder: may the source be with you!, arXiv preprint arXiv:2305.06161 (2023).

[52] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, Codegen: An open large language model for code with multi-turn program synthesis, arXiv preprint arXiv:2203.13474 (2022).

[53] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., Competition-level code generation with alphacode, Science 378 (6624) (2022) 1092–1097.

[54] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).

[55] A. Esashi, P. Lertpongrujikorn, M. A. Salehi, Action engine: An llm-based framework for automatic faas workflow generation, arXiv preprint arXiv:2411.19485 (2024).

[56] Z. Li, S. Xu, K. Mei, W. Hua, B. Rama, O. Raheja, H. Wang, H. Zhu, Y. Zhang, Autoflow: Automated workflow generation for large language model agents, arXiv preprint arXiv:2407.12821 (2024).

[57] J. Xu, W. Du, X. Liu, X. Li, Llm4workflow: An llm-based automated workflow model generation tool, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 2394–2398.

[58] M. Wang, T. Pfandzelter, T. Schirmer, D. Bermbach, Llm4faas: No-code application development using llms and faas, arXiv preprint arXiv:2502.14450 (2025).

[59] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, The VLDB Journal 33 (2) (2024) 507–541.

[60] A. Zubaroğlu, V. Atalay, Data stream clustering: a review, Artificial Intelligence Review 54 (2) (2021) 1201–1236.

[61] A. Flink, Apache flink, `https://flink.apache.org/`, accessed: 2025-04-04 (2025).

[62] A. Storm, Apache storm, `https://storm.apache.org/`, accessed: 2025-04-04 (2025).

[63] A. Spark, Apache spark, `https://spark.apache.org/`, accessed: 2025-04-04 (2025).

[64] A. Kafka, Apache kafka, `https://kafka.apache.org/`, accessed: 2025-04-04 (2025).

[65] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, K. Tzoumas, State management in apache flink®: consistent stateful distributed stream processing, Proceedings of the VLDB Endowment 10 (12) (2017) 1718–1729.

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

[67] O. Yoran, T. Wolfson, B. Bogin, U. Katz, D. Deutch, J. Berant, Answering questions by meta-reasoning over multiple chains of thought, arXiv preprint arXiv:2304.13007 (2023).

[68] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, K. Narasimhan, Tree of thoughts: Deliberate problem solving with large language models, Advances in neural information processing systems 36 (2023) 11809–11822.

[69] W. Fungwacharakorn, N. H. Thanh, M. M. Zin, K. Satoh, Layer-of-thoughts prompting (lot): Leveraging llm-based retrieval with constraint hierarchies, arXiv preprint arXiv:2410.12153 (2024).

[70] H. Huan, L. Liu, B. Huan, X. Chen, J. Zhan, Q. Liu, A theoretical investigation of modelling the temperature measurement in oil pipelines with edge devices, Measurement 168 (2021) 108440.

[71] V. Pandimurugan, J. Amudhavel, M. Sambath, et al., Hybrid compression technique for image hiding using huffman, rle and dwt, Materials Today: Proceedings 57 (2022) 2228–2233.

[72] Z. Liu, W. Lin, N. Li, D. Lee, Detecting and filtering instant messaging spam - a global and personalized approach, in: 1st IEEE ICNP Workshop on Secure Network Protocols, 2005. (NPSec)., 2005, pp. 19–24.

[73] M. AI, Models overview, `https://docs.mistral.ai/getting-started/models/models_overview/`, accessed: 2025-04-04 (2025).

[74] Groq, Models, `https://console.groq.com/docs/models`, accessed: 2025-04-04 (2025).

[75] OpenAI, Pricing, `https://platform.openai.com/docs/pricing`, accessed: 2025-04-04 (2025).

[76] Anthropic, Pricing, `https://www.anthropic.com/pricing`, accessed: 2025-04-04 (2025).

[77] D. Sun, Y. Cui, M. Wu, S. Gao, R. Buyya, An energy efficient and runtime-aware framework for distributed stream computing systems, Future Gener. Comput. Syst. 136 (2022) 252–269.

[78] H. Li, H. Dai, Z. Liu, H. Fu, Y. Zou, Dynamic energy-efficient scheduling for streaming applications in storm, Computing 104 (2) (2022) 413–432.

[79] X. Huang, Y. Jiang, H. Fan, H. Tang, Y. Wang, J. Jin, H. Wan, X. Zhao, Tata: Throughput-aware task placement in heterogeneous stream processing with deep reinforcement learning, in: 2021 IEEE BDCloud, 2021, pp. 44–54.

[80] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, et al., A survey on evaluation of large language models, ACM transactions on intelligent systems and technology 15 (3) (2024) 1–45.

[81] Z. Guo, R. Jin, C. Liu, Y. Huang, D. Shi, L. Yu, Y. Liu, J. Li, B. Xiong, D. Xiong, et al., Evaluating large language models: A comprehensive survey, arXiv preprint arXiv:2310.19736 (2023).