Discovery Agents for Real-Time Analytics: Toward Proactive Insight Systems

Gaetano Rossiello
IBM
New York City, New York, USA

Dharmashankar Subramanian
IBM
Yorktown Heights, New York, USA

Abstract

Modern analytics systems are fundamentally reactive, requiring users to define queries over increasingly complex and continuously evolving data. In real-time streaming environments, this paradigm breaks down, as the space of potential insights becomes too large to enumerate manually. We present a multi-agent architecture for autonomous insight discovery over real-time data streams. The system implements a continuous discovery loop in which agents generate hypotheses, compile them into executable analytics, validate generated artifacts, and produce visualizations and deployable applications. The architecture leverages Apache Kafka for event-driven coordination, Apache Flink for stream processing, and large language models to implement specialized agents. A key contribution is a contract-driven design based on typed intermediate artifacts, enabling modularity, observability, lineage, and safer execution of dynamically generated analytics. Through use cases in retail, finance, and public data, we show how this architecture supports a shift from query-driven analytics to proactive, discovery-driven systems.

1 Introduction

Data analytics systems have evolved from manual SQL workflows to dashboards and, more recently, LLM-powered copilots. Despite these advances, the dominant interaction model remains largely unchanged: analytics is still *query-driven*. Users formulate questions, and systems translate them into computations, visualizations, or reports. This model assumes that users know what to ask. In modern data environments, this assumption increasingly fails. Organizations operate over high-dimensional, heterogeneous, and continuously updated data sources. In particular, real-time streaming systems such as Apache Kafka [6, 11] introduce a dynamic setting in which new patterns may emerge continuously, while the space of possible analytical questions grows combinatorially. As a result, many insights remain undiscovered not because they are difficult to compute, but because no analyst explicitly formulated the corresponding hypothesis.

Recent work on LLM-powered data agents has begun to automate parts of the analytics lifecycle, including data preparation, code generation, visualization, insight generation, and reporting [5, 19, 22, 26]. Existing systems demonstrate the potential of agents for business intelligence, dashboard generation, insight management, heterogeneous data analysis, and multi-agent reasoning [1, 9, 15, 21, 23, 25]. Benchmarks for data-driven discovery and analytics agents further show that multi-step insight generation, hypothesis formulation, and robust execution are becoming central evaluation problems [7, 16, 20]. However, most existing work primarily targets batch datasets, user-specified goals, conversational analysis, or dashboard generation. Less attention has been given to how autonomous discovery agents can operate continuously over real-time streams, coordinate through production data infrastructure, and expose typed intermediate artifacts for observability, validation, and deployment.

We propose a shift from *task-driven analytics* to *discovery-driven analytics*, where analytical objectives are generated, validated, and refined by the system itself. Our approach decomposes the analytics lifecycle into specialized agents that transform data into hypotheses, executable analytics, validation reports, visualization specifications, and deployable applications. More broadly, we study the following operational model: *analytics as continuous autonomous discovery over real-time data streams.*

This paper makes three contributions. First, we formulate autonomous insight discovery as a continuous agentic workload over streaming data. Second, we propose a contract-driven architecture in which agents exchange typed artifacts for hypotheses, analytic plans, generated code, validation reports, visualization specifications, and deployment manifests. Third, we discuss representative use cases in retail, finance, and public data, highlighting early lessons for observability, validation, and human oversight in agent-driven analytics systems.

2 Related Work

LLM-powered data agents are emerging as a general abstraction for automating data management, preparation, analysis, visualization, and reporting. Recent surveys characterize these systems as tool-using agents that combine planning, retrieval, code generation, execution, and reflection across the data lifecycle [5, 19, 22, 26], while recent systems demonstrate their potential for heterogeneous data analysis, business intelligence, skill selection, external knowledge retrieval, and multi-agent reasoning [1, 9, 15, 21]. Related work on automated visualization, insight management, and data-to-dashboard generation shows how LLMs can produce charts, visual explanations, dashboard specifications, and navigable insight spaces from data and user intent [4, 8, 23–25]. In parallel, benchmarks and discovery-oriented systems increasingly frame insight generation as the target capability, evaluating agents on multi-step discovery, open-ended data analysis, business insight generation, heterogeneous data sources, and scientific hypothesis exploration [2, 7, 14, 16–18, 20]. Finally, work at the intersection of LLMs and databases highlights persistent challenges around trust, correctness, efficiency, and safe execution [10, 13]. Our work builds on these directions but differs in focus: rather than targeting static datasets, user-specified goals, reports, or interactive dashboards, we propose a streaming, contract-driven architecture in which agents continuously generate hypotheses, compile analytics, validate artifacts, produce visualizations, and package the result into deployable applications.

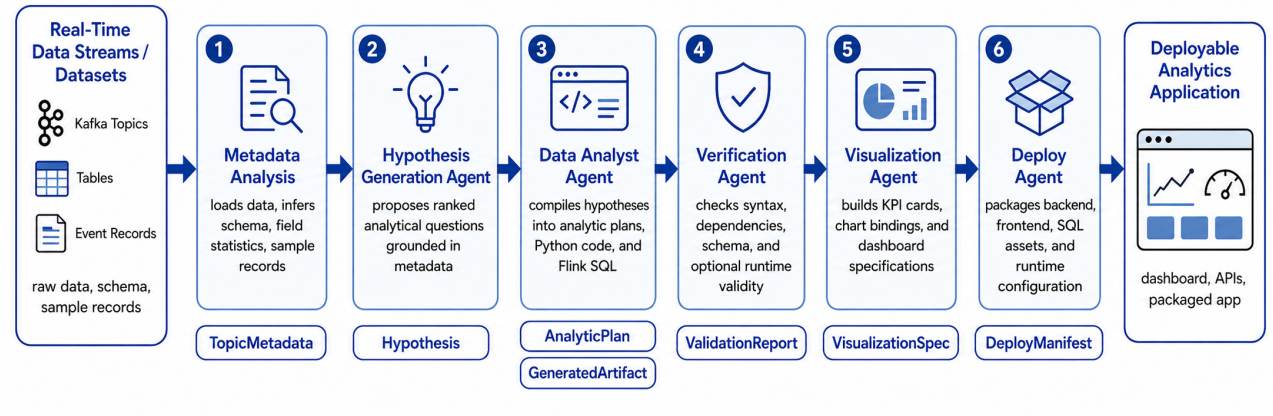

Figure 1: Contract-driven discovery-agent pipeline for real-time analytics. Real-time data streams or datasets are first converted into structured metadata, then transformed by specialized agents into ranked hypotheses, analytic plans, executable Python and FlinkSQL [3] artifacts, validation reports, visualization specifications, and finally a deployable analytics application. Each stage produces typed intermediate artifacts that support lineage, observability, verification, and human oversight.

3 Architecture: Discovery Agents over Streaming Data

The proposed system implements a multi-agent architecture for transforming real-time data streams into deployable analytics applications. At a high level, the system follows the pipeline shown in Figure 1:

$$\textit{Data} \rightarrow \textit{Metadata} \rightarrow \textit{Hypotheses} \rightarrow \textit{Analytics} \rightarrow \textit{Validation} \rightarrow \textit{Visualization} \rightarrow \textit{Application}$$

Unlike traditional data pipelines, where transformations are specified in advance, the computational workflow is constructed dynamically at runtime. Raw data is first converted into a semantic representation, `TopicMetadata`, capturing schema information, field statistics, sample records, and contextual metadata. This representation is then used to generate analytical hypotheses, which are compiled into executable artifacts, validated, visualized, and finally packaged into deployable applications.

A central design principle is the use of *typed artifact contracts*. Each agent consumes and produces explicit intermediate representations such as `Hypothesis`, `AnalyticPlan`, `GeneratedArtifact`, `ValidationReport`, `VisualizationSpec`, and `DeployManifest`. These contracts make the discovery process modular, inspectable, and reproducible.

The architecture supports both batch and streaming execution. In streaming deployments, Kafka acts as the communication backbone between agents, enabling asynchronous coordination and parallel exploration of hypotheses. Flink supports execution of generated SQL artifacts over continuous streams and provides operational visibility into the analytical pipeline. Together, these components allow the system to operate as a continuous discovery loop, where new data can trigger new hypotheses and updated analytics without manual query authoring.

4 Agent Semantics and Artifact Contracts

The system decomposes the analytics lifecycle into specialized agents, each responsible for a semantic transformation over typed artifacts. This design avoids monolithic LLM workflows and makes the discovery process observable at each stage.

4.1 Hypothesis Generation Agent

The Hypothesis Generation Agent initiates the discovery process by proposing analytical questions from data characteristics. It consumes `TopicMetadata`, including schema definitions, field statistics, sample records, and optional domain annotations.

The agent is grounded in the taxonomy of data-analysis questions proposed by Leek and Peng [12], which distinguishes descriptive, exploratory, inferential, predictive, causal, and mechanistic questions. This taxonomy is useful because different question types imply different analytical assumptions and methods: descriptive questions summarize observed data, exploratory questions search for patterns, inferential questions ask whether patterns generalize, predictive questions estimate future or unseen outcomes, causal questions ask about interventions, and mechanistic questions seek to explain underlying processes.

Before generating hypotheses, the agent categorizes fields into analytical roles, such as numerical, categorical, temporal, or textual attributes. This grounding step constrains generation to questions that can be expressed using available data operations. For example, numerical fields may support descriptive summaries or anomaly detection, categorical fields may support exploratory group comparisons, and temporal fields may support predictive trend analysis.

The output is a set of structured `Hypothesis` objects:

```
{
  "category": "descriptive",
  "question": "...",
  "rationale": "...",
  "expected_insight": "...",
  "priority": 8
}
```

Each hypothesis includes a category, rationale, expected insight, and priority score. This structure allows downstream agents to

select appropriate analytical strategies and focus computational resources on higher-value questions.

4.2 Data Analyst Agent

The Data Analyst Agent translates hypotheses into executable analytics. Given a `Hypothesis` and the corresponding `TopicMetadata`, it produces an `AnalyticPlan` and one or more `GeneratedArtifact` objects.

A key design choice is dual artifact generation. Python artifacts provide flexible batch execution and support complex analytical logic, while FlinkSQL artifacts express equivalent computations for continuous stream processing. This allows the same analytical intent to be evaluated both offline and over live data streams.

The agent first constructs an analytic plan that identifies the required fields, the intended computation, the expected output schema, and the assumptions needed to answer the hypothesis. This planning step separates *what* should be computed from *how* it is implemented, making the generated code easier to validate, debug, and regenerate when failures occur.

Generated Python artifacts follow a strict runtime contract:

```
def analyze(data_records):
    return {"results": [...]}
```

The output structure is standardized so that downstream validation, visualization, and deployment agents can consume results without relying on ad hoc parsing. Conceptually, the Data Analyst Agent acts as a compiler from analytical intent to executable programs.

This compiler-like role is important because different hypotheses require different analytical operators. Descriptive hypotheses may compile into aggregations and summaries, exploratory hypotheses into group comparisons or correlations, predictive hypotheses into trend or forecasting logic, and temporal hypotheses into windowed computations. By making these choices explicit in the `AnalyticPlan`, the system preserves a traceable link between the original question, the selected analytical method, and the generated executable artifact.

4.3 Verification Agent

The Verification Agent acts as a quality gate for generated artifacts. Since LLM-generated code may contain syntax errors, invalid assumptions, unsafe imports, or schema mismatches, validation is treated as a first-class stage rather than an optional post-processing step. This stage turns artifact correctness into an explicit, inspectable object, rather than leaving reliability as an implicit property of the generated code.

The agent performs syntax checks, dependency checks, schema validation, and, when sample data is available, runtime validation. The output is a structured `ValidationReport`:

```
{
  "status": "VALIDATED",
  "syntax_check": true,
  "schema_check": true,
  "runtime_check": false,
  "issues": []
}
```

Validation issues are classified as errors, warnings, or informational messages. Errors prevent artifacts from progressing, while warnings allow execution with caveats. Rejected artifacts can trigger regeneration with validation feedback, enabling iterative refinement.

4.4 Visualization Agent

The Visualization Agent converts validated analytical outputs into dashboard specifications. Rather than requiring manual chart design, the agent inspects output schemas and, when possible, sample execution results to infer appropriate visual encodings. It maps the semantic shape of the result—for example categorical distributions, temporal trends, scalar summaries, or ranked lists—to dashboard elements such as bar charts, line charts, KPI cards, tables, and summary views.

The output is a `VisualizationSpec` describing charts, KPI cards, data bindings, and layout:

```
{
  "charts": [
    {
      "type": "bar",
      "x": "category",
      "y": "count"
    }
  ],
  "kpis": [...]
}
```

This specification separates analytical computation from presentation. It also enables multiple frontends to render the same discovered insight while preserving a traceable connection to the originating hypothesis and generated artifact.

4.5 Deploy Agent

The Deploy Agent materializes the validated analytics and visualization specifications into a runnable application. It packages generated code, dashboard configuration, backend services, frontend components, and deployment metadata. It also preserves the connection between runtime assets and upstream artifacts, so that deployed dashboards can be traced back to the hypotheses, code, validation reports, and visualization specifications that produced them.

The output is a `DeployManifest` that records generated files, configuration parameters, artifact lineage, and deployment instructions. This manifest serves as the final contract between the discovery pipeline and the runtime environment. In this way, the system does not merely generate insights; it packages them into inspectable applications that can be executed, reviewed, and extended by users.

4.6 Contract-Driven Coordination

Typed artifact contracts provide the coordination layer across agents. Each transformation is explicit:

$$\texttt{TopicMetadata} \rightarrow \texttt{Hypothesis} \rightarrow \texttt{AnalyticPlan} \rightarrow \texttt{GeneratedArtifact} \rightarrow \texttt{ValidationReport} \rightarrow \texttt{VisualizationSpec} \rightarrow \texttt{DeployManifest}$$

This structure enables lineage tracking across the discovery process. Hypotheses reference their source topics, generated artifacts

reference the hypotheses they implement, validation reports reference the artifacts they evaluate, and deployment manifests reference all upstream objects.

In streaming deployments, these artifacts can be exchanged through Kafka topics, allowing multiple hypotheses to be explored in parallel. This creates a natural execution model for agentic analytics: agents operate asynchronously, artifacts provide the shared state, and validation reports control progression through the pipeline. The same contract structure also creates intervention points for human oversight. Users can review hypotheses before execution, inspect generated code before deployment, examine validation reports, or adjust visualization specifications. This balances automation with control and makes the system suitable for production-oriented analytics settings.

5 Use Cases and Early Lessons

Given the exploratory nature of the system, we evaluate it through representative use cases rather than a formal benchmark. The goal is to characterize the types of discovery workflows enabled by the architecture and identify early design lessons.

5.1 Retail Analytics

In retail analytics, the system processes transaction streams containing product, category, and customer interaction data. The Hypothesis Generation Agent identifies candidate patterns such as category dominance, temporal purchasing trends, basket composition, and anomalies in sales distribution. The system then generates aggregation queries, time-series analyses, and dashboards highlighting indicators such as top-selling categories, changes in product mix, or distribution shifts. This illustrates how proactive hypothesis generation can reduce the manual effort required to design exploratory dashboards.

5.2 Financial Monitoring

In financial monitoring, the system operates over transaction or market-event streams and proposes hypotheses related to volume spikes, unusual activity distributions, deviations from expected behavior, or temporal concentration of events. Generated analytics can surface candidate anomalies and summarize them through interpretable dashboards. Continuous hypothesis generation is particularly useful in this setting because relevant patterns may evolve rapidly and may not be known in advance.

5.3 Governance and Public Data

In public data scenarios, such as NYC Open Data, the system operates over heterogeneous datasets related to public services, infrastructure, complaints, or demographic indicators. It can propose hypotheses about temporal trends, geographic differences, or relationships between service requests and neighborhoods. The resulting dashboards can help analysts and policymakers explore datasets without first defining a fixed set of queries. This use case highlights the value of autonomous discovery in broad, heterogeneous data environments where the relevant analytical questions may not be obvious upfront.

5.4 Cross-Domain Lessons

Across these scenarios, three lessons emerge. First, hypothesis generation must be grounded in metadata and field statistics to avoid irrelevant or non-executable questions. Second, validation must be treated as a first-class stage, since generated code and queries may fail syntactically, semantically, or operationally. Third, observability over intermediate artifacts is essential: users and operators need to inspect not only final dashboards, but also the hypotheses, plans, code, and validation reports that produced them. These lessons suggest that agentic analytics systems require more than LLM prompting. They require system-level support for contracts, lineage, verification, and human oversight.

6 Position and Open Challenges

We argue that proactive insight discovery should become a first-class workload for agent-first data systems. The proposed architecture shifts analytics from a query-driven process, where goals are specified externally, to a discovery-driven process, where agents generate and prioritize analytical objectives directly from data. This elevates autonomy from execution to *problem formulation*. Instead of acting only as a query processor, the system operates as a continuous discovery engine. Analytical objectives emerge from the interaction between data, metadata, validation feedback, and specialized agents. The contract-driven design is a key enabler of this transition. Typed artifacts make LLM-based generation more controllable by exposing intermediate representations that can be validated, logged, inspected, and replayed. This is especially important in streaming environments, where agent actions may be triggered continuously and must remain auditable.

Several open challenges remain. First, hypothesis quality is difficult to evaluate: a valid hypothesis may still be uninteresting, redundant, or misleading. Second, generated analytics must be checked not only for syntax, but also for statistical validity and operational safety. Third, continuous discovery may produce many candidate insights, requiring ranking, deduplication, and human-in-the-loop review. Finally, deployment raises governance questions around provenance, access control, and accountability for system-generated conclusions. These challenges suggest a broader research agenda for agentic data systems: building infrastructure that supports not only faster query answering, but also reliable, observable, and steerable autonomous exploration.

7 Conclusion

We presented a multi-agent architecture for autonomous discovery over real-time data streams. By combining Kafka-based coordination, Flink-based stream processing, and LLM-powered agents with typed artifact contracts, the system transforms data into hypotheses, analytics, validation reports, visualizations, and deployable applications. The central contribution is a shift from automating analysis to automating discovery. Rather than accelerating only user-specified workflows, the proposed architecture treats insight generation itself as a continuous, agentic process embedded within the data infrastructure.

A Generated Artifacts: NYC Parks Events Demo

This appendix summarizes the artifacts generated by the discovery-agent pipeline for the NYC Parks Events dataset [1]. The goal is not to document every generated file exhaustively, but to show how the proposed architecture materializes the discovery loop into inspectable, executable, and deployable artifacts.

A.1 Demo Summary

The NYC Parks Events Listing dataset contains public event records across New York City parks. Each record includes temporal, geographic, textual, categorical, and cost-related attributes. The pipeline processed the dataset through the full discovery loop:

`Metadata` → `Hypotheses` → `Analytics` → `Validation` → `Visualization` → `Deployment`

Table 1 summarizes the input data and generated outputs.

A.2 End-to-End Lineage Example

To illustrate the full discovery pipeline, we trace a single high-priority hypothesis through all transformation stages. This example shows how typed artifact contracts preserve lineage from an analytical question to the deployed application.

A.2.1 Stage 1: Hypothesis Generation. The Hypothesis Generation Agent produced the following hypothesis from dataset metadata:


```
{
  "id": "hyp_20260518_133512_090881",
  "created_at": "2026-05-18T13:35:12.090893",
  "source_topic_id": "topic_20260518_133450_224571",
  "category": "descriptive",
  "question": "What is the temporal distribution of NYC
    Parks events across years, months, and days of
    the week, and are there clear seasonal patterns
    in event scheduling?",
  "rationale": "With 74,880 events spanning multiple
    years, understanding when events are concentrated
    reveals how NYC Parks allocates programming
    resources across seasons.",
  "expected_insight": "We expect to find strong
    seasonality with peaks in spring/summer months
    and troughs in winter, with weekends having
    significantly more events than weekdays.",
  "priority": 9
}
```


The hypothesis references the source topic and includes a priority score used to rank candidate analyses.

A.2.2 Stage 2: Analytic Plan Generation. The Data Analyst Agent translated the hypothesis into an executable plan:


```
{
  "id": "plan_20260518_133551_981283",
  "hypothesis_id": "hyp_20260518_133512_090881",
  "approach": "Parse the date field from each event
    record to extract year, month, and day of week.
    Aggregate event counts by these temporal
    dimensions to reveal distribution patterns.",
  "steps": [
    "Load and validate data",
    "Parse date strings into datetime objects",
    "Extract year, month, and day_of_week",
    "Aggregate event counts by temporal dimensions",
    "Compute seasonal distribution",
    "Calculate weekday versus weekend statistics",
    "Return structured results"
  ],
  "output_schema": {
    "results": "dict with keys: [summary,
      yearly_distribution, monthly_distribution,
      day_of_week_distribution, seasonal_distribution,
      weekday_weekend_comparison]"
  }
}
```


The plan separates analytical intent from implementation. This makes the generated code easier to validate, debug, regenerate, and explain.

A.2.3 Stage 3: Python Artifact Generation. The agent generated a Python artifact implementing the analytic plan:


```
{
  "id": "artifact_py_20260518_133551_981378",
  "hypothesis_id": "hyp_20260518_133512_090881",
  "analytic_plan_id": "plan_20260518_133551_981283",
  "artifact_type": "python_code",
  "language": "python",
  "dependencies": ["pandas", "numpy"]
}
```


The generated function follows the standard runtime contract: it receives records as input and returns a structured result object.

Listing 1: Generated Python artifact for temporal distribution analysis.

```
def analyze_temporal_distribution(
    data_records: List[Dict[str, Any]]
) -> Dict[str, Any]:
    """Analyze temporal distribution of NYC Parks events.
       """
    df = pd.DataFrame(data_records)

    # Parse dates and keep valid records.
    df["parsed_date"] = df["date"].apply(parse_date)
    df_valid = df[df["parsed_date"].notna()].copy()

    # Derive temporal features.
    df_valid["year"] = df_valid["parsed_date"].dt.year
    df_valid["month"] = df_valid["parsed_date"].dt.month
    df_valid["day_of_week"] = df_valid["parsed_date"].dt.
        dayofweek
    df_valid["season"] = df_valid["month"].apply(get_
        season)

    # Compute distributions.
    yearly_distribution = (
        df_valid.groupby("year")
        .size()
        .reset_index(name="event_count")
    )
```

[1] https://data.cityofnewyork.us/City-Government/NYC-Parks-Events-Listing-Event-Listing/fudw-fgrp/about_data

Table 1: Summary of the NYC Parks Events discovery demo.

Component	Description
Dataset	NYC Parks Events Listing dataset, containing scheduled public events across parks in New York City.
Representative fields	`date`, `start_time`, `end_time`, `location`, `borough`, `park_name`, `title`, `description`, `event_type`, `cost_free`, `must_see`.
Generated hypotheses	Analytical questions covering descriptive, exploratory, inferential, and predictive categories.
Generated analytics	Python artifacts for batch analysis and FlinkSQL artifacts for stream-oriented execution.
Validation outputs	Structured validation reports covering syntax, dependency, schema, and runtime checks.
Generated interface	Dashboard with KPI cards, hypothesis tabs, and interactive visualizations.
Deployment output	Containerized application with backend API, frontend dashboard, source data, generated artifacts, and deployment manifest.

```
monthly_distribution = (
    df_valid.groupby("month")
    .size()
    .reset_index(name="event_count")
)

seasonal_distribution = (
    df_valid.groupby("season")
    .size()
    .reset_index(name="event_count")
)

return {
    "results": {
        "summary": {...},
        "yearly_distribution": yearly_distribution,
        "monthly_distribution": monthly_distribution,
        "seasonal_distribution": seasonal_
            distribution
    }
}
```

The key property is that the artifact does not return free-form text. It returns a structured object that can be validated, cached, visualized, and served by the generated application.

A.2.4 Stage 4: FlinkSQL Artifact Generation. The agent also generated a FlinkSQL artifact for stream-oriented execution of the same analytical intent:

```
{
  "id": "artifact_sql_20260518_133551_981443",
  "hypothesis_id": "hyp_20260518_133512_090881",
  "analytic_plan_id": "plan_20260518_133551_981283",
  "artifact_type": "flink_sql"
}
```

Listing 2: Generated FlinkSQL artifact for temporal aggregation.

```
SELECT
    EXTRACT(YEAR FROM TO_DATE(`date`, 'MM/dd/yyyy'))
        AS event_year,
    EXTRACT(MONTH FROM TO_DATE(`date`, 'MM/dd/yyyy'))
        AS event_month,
    CASE
        WHEN EXTRACT(MONTH FROM TO_DATE(`date`,
            'MM/dd/yyyy')) IN (12, 1, 2) THEN 'Winter'
        WHEN EXTRACT(MONTH FROM TO_DATE(`date`,
            'MM/dd/yyyy')) IN (3, 4, 5) THEN 'Spring'
        WHEN EXTRACT(MONTH FROM TO_DATE(`date`,
            'MM/dd/yyyy')) IN (6, 7, 8) THEN 'Summer'
        ELSE 'Fall'
    END AS season,
    COUNT(*) AS event_count
FROM `NYC_Parks_Events_Listing_-_Event_Listing`
WHERE `date` IS NOT NULL AND `date` <> ''
GROUP BY event_year, event_month, season;
```

This dual-artifact pattern is central to the architecture: Python supports flexible batch execution, while FlinkSQL provides a path toward continuous analytics over streams.

A.2.5 Stage 5: Validation. The Verification Agent validated the generated artifact before it was passed to visualization and deployment:

```
{
  "id": "validation_20260518_133936_698650",
  "artifact_id": "artifact_py_20260518_133551_981378",
  "status": "validated",
  "syntax_check": true,
  "import_check": true,
  "schema_check": true,
  "runtime_check": false,
  "issues": [],
  "feedback": "Validation passed successfully"
}
```

The validation report confirms that the generated artifact passes syntax, dependency, and schema checks. Failed validations can trigger regeneration with structured feedback, making validation part of the agentic workflow rather than a manual post-processing step.

A.2.6 Stage 6: Visualization Specification. The Visualization Agent mapped the validated output to dashboard components. Each component references a specific artifact and output path, preserving traceability from visual element to computation.

```
{
  "id": "viz_20260518_133953_921433",
  "hypothesis_ids": ["hyp_20260518_133512_090881", ...],
  "artifact_ids": [
```

```
    "artifact_py_20260518_133551_981378",
    "artifact_sql_20260518_133551_981443",
    ...
  ],
  "dashboard": {
    "kpi_cards": [
      {
        "id": "kpi_1",
        "title": "Total Events Analyzed",
       "value_field": "artifacts.artifact_py_20260518_
          133551_981378.summary.total_events_analyzed"
      }
    ],
    "charts": [
      {
        "id": "chart_1",
        "title": "Yearly Event Distribution",
        "chart_type": "bar",
       "data_source": "artifacts.artifact_py_20260518_
          133551_981378.yearly_distribution",
        "x_field": "year",
        "y_field": "event_count"
      },
      {
        "id": "chart_2",
        "title": "Monthly Event Distribution",
        "chart_type": "line",
       "data_source": "artifacts.artifact_py_20260518_
          133551_981378.monthly_distribution"
      }
    ]
  }
}
```


Figure 2 shows the generated interface.

A.2.7 Stage 7: Deployment Manifest. Finally, the Deploy Agent packaged the generated artifacts into a deployable application:


```
{
  "id": "deploy_20260518_133954_020895",
  "source_topic_id": "topic_20260518_133450_224571",
 "hypothesis_ids": ["hyp_20260518_133512_090881", ...],
  "artifact_ids": [
    "artifact_py_20260518_133551_981378",
    "artifact_sql_20260518_133551_981443",
    ...
  ],
 "visualization_spec_id": "viz_20260518_133953_921433",
  "app_name": "nyc-park-events-demo",
  "generated_files": [
    {
      "path": "backend/app/analytics/
        artifact_py_20260518_133551_981378.py",
      "type": "python",
      "description": "Python analysis for temporal
        distribution hypothesis"
    },
    {
      "path": "backend/app/sql/
        artifact_sql_20260518_133551_981443.sql",
      "type": "sql",
      "description": "FlinkSQL for temporal
        distribution hypothesis"
    }
  ]
}
```


The deployment manifest preserves the complete lineage chain. Any deployed dashboard element can be traced back through visualization specifications, validation reports, generated code, analytic plans, the originating hypothesis, and the source data.

A.3 Additional Generated Hypotheses

Table 2 summarizes additional high-priority hypotheses generated for the demo. These hypotheses follow the same transformation pipeline illustrated above.

A.4 Generated Application Structure

The Deploy Agent packages the generated artifacts into a runnable application. The resulting directory contains the backend service, frontend dashboard, source data, generated analytics, SQL artifacts, validation reports, and deployment metadata.

```
nyc-park-events-demo/
 README.md
 docker-compose.yml
 deploy_manifest.json
 VERIFICATION_REPORT.txt
 backend/
    Dockerfile
    requirements.txt
    app/
        main.py
        analytics/
           artifact_py_*.py
        sql/
            artifact_sql_*.sql
            manifest.json
 frontend/
    Dockerfile
    nginx.conf
    public/
       index.html
    src/
        dashboard.js
        styles.css
 data/
     full_dataset.csv
```

The backend exposes REST endpoints for health checks, dataset summaries, KPI values, chart data, data previews, and hypothesis metadata. The frontend consumes these endpoints and renders the generated dashboard.

A.5 Execution and Discovered Insights

The application can be launched with:

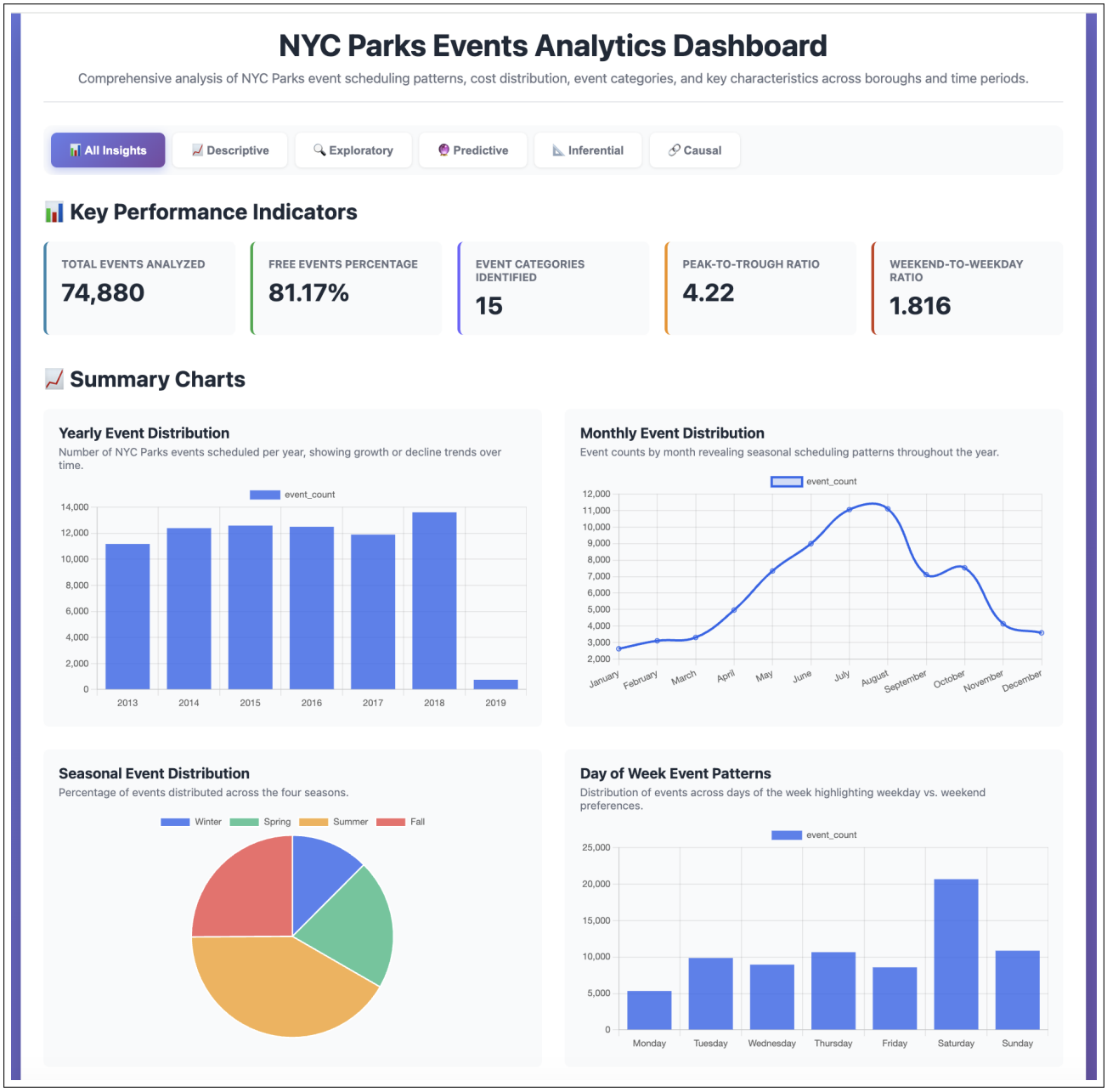

Figure 2: Generated dashboard for NYC Parks Events analytics. The interface includes KPI cards, hypothesis tabs, and automatically selected visualizations derived from validated analytical outputs.

```
docker-compose up --build
```

This starts the backend and frontend services, loads the dataset, executes the generated analytical functions, caches results, and serves them through the dashboard.

The generated analysis surfaced several representative insights:

- **Temporal patterns**: Events are concentrated in warmer months, with stronger activity during summer.
- **Cost distribution**: Most events are free, with variation across event types and locations.
- **Event categories**: Frequent categories include fitness, cultural programming, nature education, and family activities.
- **Weekend preference**: Events are more concentrated on weekends than weekdays.
- **Borough distribution**: A large share of events is concentrated in Manhattan and Brooklyn.

Table 2: Additional representative hypotheses generated for the NYC Parks Events dataset.

ID	Category	Question
H2	Exploratory	What proportion of events are free versus paid, and how does this ratio vary by event type, location, and time of year?
H3	Descriptive	What are the most common event categories, and how are they distributed across parks and boroughs?
H4	Inferential	Is there a relationship between an event being marked as `must_see` and characteristics such as cost, location, time of day, or event type?
H5	Predictive	Can event duration be predicted from event type, location, day of week, and whether the event is free?

These results illustrate the intended behavior of the system: the user does not manually write the analytical queries. Instead, the system proposes hypotheses, generates analytics, validates outputs, and packages the results into an application.

A.6 Lessons from the Demo

The NYC Parks Events demo highlights five practical lessons.

(1) **Typed contracts improve reliability.** Explicit intermediate artifacts make the discovery process easier to inspect, validate, and replay.
(2) **Hypothesis generation must be grounded.** Metadata and field statistics are necessary to avoid irrelevant or non-executable questions.
(3) **Validation is essential.** Generated code and SQL must be checked before they are used in downstream visualizations or deployed applications.
(4) **Dual artifacts support multiple execution modes.** Python artifacts support batch analytics, while FlinkSQL artifacts provide a path toward streaming execution.
(5) **Lineage enables oversight.** Deployment manifests preserve the connection between hypotheses, code, validation reports, visualizations, and application files.

Overall, the demo shows that the proposed architecture does more than generate isolated insights. It produces a complete, inspectable chain from dataset metadata to a deployable analytics application.

References

[1] Amirhossein Abaskohi, Amrutha Varshini Ramesh, Shailesh Nanisetty, Chirag Goel, et al. 2025. AgentAda: Skill-Adaptive Data Analytics for Tailored Insight Discovery. *arXiv preprint arXiv:2504.07421* (2025).
[2] Dhruv Agarwal, Bodhisattwa Prasad Majumder, Reece Adamson, Megha Chakravorty, et al. 2025. AutoDiscovery: Open-ended Scientific Discovery via Bayesian Surprise. *arXiv preprint arXiv:2507.00310* (2025).
[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38, 4 (2015).
[4] Zichen Chen, Jiefeng Chen, Sercan O. Arik, Misha Sra, Tomas Pfister, and Jinsung Yoon. 2025. CoDA: Agentic Systems for Collaborative Data Visualization. *arXiv preprint arXiv:2510.03194* (2025).
[5] Yanjie Fu, Dongjie Wang, Wangyang Ying, Xiangliang Zhang, Huan Liu, and Jian Pei. 2025. Autonomous Data Agents: A New Opportunity for Smart Data. *arXiv preprint arXiv:2509.18710* (2025).
[6] Nishant Garg. 2013. *Apache Kafka.* Packt Publishing.
[7] Ken Gu, Ruoxi Shang, Ruien Jiang, Keying Kuang, et al. 2024. BLADE: Benchmarking Language Model Agents for Data-Driven Science. In *Findings of the Association for Computational Linguistics: EMNLP 2024.*
[8] Enamul Hoque and Mohammed Saidul Islam. 2025. Natural Language Generation for Visualizations: State of the Art, Challenges and Future Directions. *Computer Graphics Forum* (2025).
[9] Jie Jiang, Haining Xie, Siqi Shen, Yu Shen, Zihan Zhang, Meng Lei, Yifeng Zheng, Yang Li, Chunyou Li, Danqing Huang, et al. 2025. SiriusBI: A Comprehensive LLM-Powered Solution for Data Analytics in Business Intelligence. *Proceedings of the VLDB Endowment* 18, 12 (2025), 4860–4873.
[10] Kyoungmin Kim and Anastasia Ailamaki. 2024. Trustworthy and Efficient LLMs Meet Databases. *arXiv preprint arXiv:2412.18022* (2024).
[11] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB.* Athens, Greece.
[12] Jeffery T. Leek and Roger D. Peng. 2015. What is the question? *Science* 347, 6228 (2015), 1314–1315.
[13] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proceedings of the VLDB Endowment* (2024). doi:10.14778/3685800.3685838
[14] Shicheng Liu, Yucheng Jiang, Sajid Farook, Camila Nicollier Sanchez, David Fernando Castro Pena, and Monica S. Lam. 2026. DataSTORM: Deep Research on Large-Scale Databases using Exploratory Data Analysis and Data Storytelling. *arXiv preprint arXiv:2604.06474* (2026).
[15] Xiaochuan Liu, Yuanfeng Song, Xiaoming Yin, and Xing Chen. 2025. DataSage: Multi-agent Collaboration for Insight Discovery with External Knowledge Retrieval, Multi-role Debating, and Multi-path Reasoning. *arXiv preprint arXiv:2511.14299* (2025).
[16] Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, et al. 2025. DiscoveryBench: Towards Data-Driven Discovery with Large Language Models. In *The Thirteenth International Conference on Learning Representations.*
[17] Bodhisattwa Prasad Majumder, Harshit Surana, Sanchaita Hazra, Ashish Sabharwal, and Peter Clark. 2024. Data-driven Discovery with Large Generative Models. In *International Conference on Machine Learning.*
[18] Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, et al. 2025. Kosmos: An AI Scientist for Autonomous Discovery. *arXiv preprint arXiv:2511.02824* (2025).
[19] Mizanur Rahman, Amran Bhuiyan, Mohammed Saidul Islam, Md Tahmid Rahman Laskar, Ridwan Mahbub, Ahmed Masry, Shafiq Joty, and Enamul Hoque. 2025. LLM-Based Data Science Agents: A Survey of Capabilities, Challenges, and Future Directions. *arXiv preprint arXiv:2510.04023* (2025).
[20] Gaurav Sahu, Abhay Puri, Juan Rodriguez, Amirhossein Abaskohi, et al. 2024. InsightBench: Evaluating Business Analytics Agents Through Multi-Step Insight Generation. *arXiv preprint arXiv:2407.06423* (2024).
[21] Ji Sun, Guoliang Li, Peiyao Zhou, Yihui Ma, Jingzhe Xu, and Yuan Li. 2025. AgenticData: An Agentic Data Analytics System for Heterogeneous Data. *arXiv preprint arXiv:2508.05002* (2025).
[22] Maojun Sun, Ruijian Han, Binyan Jiang, Houduo Qi, Defeng Sun, Yancheng Yuan, and Jian Huang. 2025. A Survey on Large Language Model-based Agents for Statistics and Data Science. *The American Statistician* (2025). doi:10.1080/00031305.2025.2561140
[23] Luoxuan Weng, Xingbo Wang, Junyu Lu, Yingchaojie Feng, Yihan Liu, Haozhe Feng, Danqing Huang, and Wei Chen. 2025. InsightLens: Augmenting LLM-Powered Data Analysis with Interactive Insight Management and Navigation. *IEEE Transactions on Visualization and Computer Graphics* (2025).
[24] Yang Wu, Yao Wan, Hongyu Zhang, Yulei Sui, Wucai Wei, Wei Zhao, Guandong Xu, and Hai Jin. 2024. Automated Data Visualization from Natural Language via Large Language Models: An Exploratory Study. *Proceedings of the ACM on Management of Data* 2, 3, Article 115 (2024).
[25] Ran Zhang and Mohannad Elhamod. 2025. Data-to-Dashboard: Multi-Agent LLM Framework for Insightful Visualization in Enterprise Analytics. *arXiv preprint arXiv:2505.23695* (2025).
[26] Yizhang Zhu, Liangwei Wang, Chenyu Yang, Xiaotian Lin, Boyan Li, et al. 2025. A Survey of Data Agents: Emerging Paradigm or Overstated Hype? *arXiv preprint arXiv:2510.23587* (2025).