LLM-Powered Conversational Analytics for Software Project Management: An Empirical Study

Rodrigo A. Vilar
Joaquim Honório
rodrigo@virtus.ufcg.edu.br
joaquim.honorio@virtus.ufcg.edu.br
Intelligent Software Engineering Group - VIRTUS/UFCG
Campina Grande, PB, Brazil

Felipe Ramos
felipe.ramos@ifpb.edu.br
Federal Institute of Paraíba (IFPB)
Santa Luzia, PB, Brazil
Intelligent Software Engineering Group - VIRTUS/UFCG
Campina Grande, PB, Brazil

Mirko Perkusich
Danyllo Albuquerque
mirko@virtus.ufcg.edu.br
danyllo.albuquerque@virtus.ufcg.edu.br
Intelligent Software Engineering Group - VIRTUS/UFCG
Campina Grande, PB, Brazil

Evandro Costa
evandro@ic.ufal.br
Federal University of Alagoas (UFAL)
Maceió, AL, Brazil

Danilo F. S. Santos
Angelo Perkusich
danilo.santos@dee.ufcg.edu.br
perkusic@virtus.ufcg.edu.br
Intelligent Software Engineering Group - VIRTUS/UFCG
Campina Grande, PB, Brazil

Abstract

Modern software projects generate large volumes of heterogeneous data that must be continuously analyzed to support timely decision-making. However, traditional software analytics relies on manually defined queries and specialist-maintained dashboards, limiting flexibility and accessibility for practitioners. This paper presents an empirical study of a conversational analytics approach that enables practitioners to query and visualize project data using natural language. The solution integrates large language models with a CubeJS semantic analytics layer, orchestrated through a low-code Node-RED backend. The pipeline rewrites incomplete questions, selects relevant analytical contexts using semantic similarity, and generates executable CubeJS JSON queries. We evaluate the approach in an industrial setting, using 15 analytical cubes built over an industrial database containing data from 463 software projects. The results show an execution accuracy of 73.4% (95% Wilson CI [68.1%, 78.0%]) and a mean end-to-end latency of 15.36 s, reduced by 71.7% via semantic caching. A Text-to-SQL baseline achieves 62.08% (95% CI [56.5%, 67.4%]); the non-overlapping confidence intervals provide evidence supporting the advantage of grounding query generation in a semantic analytics layer. These findings suggest that LLM-driven conversational analytics can broaden access to software project insights in industrial settings.

Keywords

Software Analytics, Large Language Models, Software Project Management

1 Introduction

Modern software projects generate large volumes of heterogeneous and continuously evolving data, including issue tracking records, sprint boards, and continuous integration logs. Analyzing such data is essential to support decision-making, improve software quality, and enable data-driven management practices [4, 20]. In a large-scale industrial setting, hundreds of software projects may generate data across hundreds of tables and views, resulting in complex and fragmented data landscapes. However, traditional software analytics pipelines depend on manually crafted queries and specialist-maintained dashboards. These static approaches often fail to keep pace with the rapid, iterative feedback cycles required in modern software development. Furthermore, non-technical stakeholders frequently struggle to obtain insights, and delays hinder iterative planning.

Software analytics aims to enable software practitioners to explore and analyze artifacts, obtaining insightful and actionable information for tasks related to software systems, users, and the development process [28]. The StackMine project at Microsoft, one of the first industrial software analytics systems, demonstrated that analytics can have a positive impact on practice. It emphasized focusing on problems that practitioners care about, leveraging domain knowledge to interpret data, building prototypes early to gather feedback, and evaluating results using criteria aligned with real-world tasks [29].

Despite growing interest, adoption remains challenging. A case study on software analytics adoption for managerial decision-making found that organizations use software engineering data to support decision-making; however, the usefulness of analytics tools depends on the alignment between the information they provide and practitioners' actual needs [22]. The study identified 19 use cases mapped to ISO/IEC/IEEE 12207:2017 processes and highlighted project-, human-, and context-specific factors that affect adoption. Figalist et al. [11] conducted an exploratory case study and identified a vicious cycle that prevents practitioners from adopting software analytics and business intelligence (BI) solutions, consisting of four key drivers: (i) inability to prove value, (ii) lack of priority, time, and resources, (iii) low data quality and inability to bridge the cultural gap, and (iv) ineffective prototypical analysis.

Furthermore, a systematic literature review found that most software analytics research primarily targets developers and uses only one artifact, thereby limiting cross-artifact insights and suggesting that the field is still in its embryonic stage [1]. Meanwhile, studies

that integrate software quality models into analytics tools report a positive reception by practitioners but underscore the need to connect analytics outputs to higher-level quality goals [18]. Other work shows that tuning data-mining algorithms can dramatically improve defect prediction performance: a differential evolution tuner increased detection precision from 0% to 60% with relatively few attempts [12]. These findings suggest that while software analytics holds promise, existing tools often lack the flexibility and responsiveness needed for modern development environments.

The lack of flexibility and responsiveness in current analytics tools highlights the need for more adaptive solutions. Large language models (LLMs) and natural language interfaces offer a promising route to democratize software analytics. Recent systems, such as AllHands, leverage GPT-4 to provide "ask-me-anything" analytics over large feedback datasets, achieving high ratings in accuracy, comprehensiveness, and readability [27]. GateLens combines LLM reasoning with domain-specific relational modeling to achieve high benchmark accuracy and significantly reduce release-validation time [15]. Despite these successes, evaluations also reveal limitations: Tufano et al. [25] cataloged more than 40 conversational use cases for ChatGPT but reported issues such as hallucination and outdated knowledge; Arulmohan et al. [3] showed that GPT-3.5 lags behind a Conditional Random Field model on domain-model extraction; and Abedu et al. [2] demonstrated that an off-the-shelf repository chatbot answers only a small fraction of user queries correctly. Systematic reviews further highlight challenges in semantic understanding, context assembly, and domain adaptation when applying LLMs to software engineering [13, 14, 31].

This paper presents an empirical study of a conversational analytics approach for software project management, integrating a large language model (LLaMA-3.1-8B) with a CubeJS semantic analytics layer, orchestrated via a low-code Node-RED backend. The proposed approach combines rule-based NLP, LLM-driven question rewriting, and semantic similarity matching to translate natural-language questions into executable CubeJS queries.

We address two research questions (RQs):

RQ1 *How reliably can a hybrid conversational pipeline translate natural-language questions about software project data into executable CubeJS JSON plans?*

RQ2 *What responsiveness does the pipeline achieve, and how effectively do semantic caching and CubeJS pre-aggregations reduce end-to-end latency?*

By answering these RQs, we make the following contributions:

(1) We empirically evaluate a hybrid conversational analytics pipeline that combines rule-based NLP with LLM-driven question rewriting and semantic similarity matching. The pipeline is deployed in a low-code environment (Node-RED) and integrates with the CubeJS analytics layer.
(2) We introduce a semantic cache and a similarity matching mechanism that select relevant analytic contexts and reduce redundant computation, enabling interactive response times.
(3) We conduct an empirical study in an industrial setting with 463 projects and 15 analytical cubes, evaluating query generation accuracy, latency, and cache effectiveness. We further compare against a Text-to-SQL baseline fine-tuned specifically for SQL generation, showing that grounding generation in a semantic analytics layer improves translation accuracy.

2 Related Work

This work sits at the intersection of four areas: (i) software analytics and its adoption in practice, (ii) LLM-powered conversational analytics, (iii) low-code architectures, and (iv) text-to-JSON generation and schema mapping. We summarize each area and position our contribution.

Software analytics and adoption. Prior efforts, including Microsoft's StackMine [29] and adoption studies by Rique et al. [22] and Figalist et al. [11], demonstrate the potential of analytics for managerial decision-making while highlighting challenges such as developer-centric focus, limited integration across artifacts, and misalignment with practitioners' needs. In practice, these approaches typically rely on predefined dashboards and manually crafted queries, which restrict exploratory analysis and delay access to insights. In contrast, our work advances this line by integrating diverse artifacts via CubeJS and enabling managers to obtain real-time indicators without relying on BI experts.

LLM-powered analytics. Conversational systems such as AllHands [27] and GateLens [15] illustrate the potential of LLMs for software analytics. However, empirical studies report some limitations, including hallucinations, stale knowledge, and low domain accuracy [2, 3, 25]. These issues become particularly critical in scenarios requiring structured outputs, such as query generation, where syntactic correctness does not guarantee semantic validity. Unlike fully LLM-based approaches, our method constrains the role of LLMs to question rewriting and explanation, while delegating query translation to a rule-based SpaCy parser, yielding stable accuracy with lower maintenance.

Low-code architectures. Node-RED is widely used for IoT and IIoT orchestration [5, 6, 10, 16, 21, 23, 24], while CubeJS provides an on-premise semantic layer with caching and query APIs. Although both technologies are well established, their use in conversational analytics remains underexplored. To the best of our knowledge, no prior work has combined these tools to support conversational query generation over a semantic analytics layer. This integration enables rapid prototyping, modular pipeline composition, and a clear separation of concerns between orchestration and semantic validation, which are essential for building reliable conversational analytics systems in industrial settings.

Text-to-JSON generation and schema mapping. Recent work on LLM-based JSON generation (e.g., Escarda-Fernández et al. [9], Chen et al. [7], Neubauer et al. [19]) shows promise but also highlights key limitations, including output inconsistency, sensitivity to prompt design, and limited guarantees of semantic correctness. These challenges are consistent with findings from the Text-to-SQL literature, which emphasize that reliable natural language-to-query translation requires strong schema grounding and explicit validation mechanisms [17].

Despite recent advances, many approaches still rely on end-to-end LLM generation driven by prompt engineering, offering limited control over structural correctness and schema alignment. As highlighted in the Text-to-SQL literature, robust solutions typically

require modular designs with schema grounding, intermediate representations, and validation mechanisms to ensure reliable query generation [17].

In contrast, we adopt and empirically evaluate a hybrid design that separates semantic interpretation from structured query generation. By combining deterministic linguistic parsing (SpaCy) with constrained LLM usage for rewriting, and grounding both in a semantic analytics layer (CubeJS), our approach introduces explicit safeguards that improve schema alignment and reduce semantically incorrect outputs. We provide empirical evidence for this architectural contrast through a controlled comparison between the hybrid pipeline and a direct Text-to-SQL baseline, showing that grounding query generation in a semantic analytics layer improves translation accuracy over end-to-end LLM-driven generation.

3 Motivating Example

This section presents a motivating example that illustrates common challenges in software analytics for project management.

3.1 Industrial Context

Consider a software organization that coordinates dozens of software projects annually across domains such as web and mobile systems, artificial intelligence, embedded systems, and hardware design. These projects are typically executed by multidisciplinary teams following agile methodologies such as Scrum or Kanban.

To support development and management activities, the organization maintains an internal platform that integrates multiple tools, including issue tracking systems, version control repositories, testing frameworks, and continuous integration pipelines. This platform ensures traceability across artifacts and supports project monitoring at different levels of abstraction.

Despite this integrated infrastructure, project leads and managers face a persistent challenge: key decision-making data remains fragmented across heterogeneous sources. As a result, obtaining timely and actionable insights requires significant manual effort.

3.2 Challenges in Traditional Software Analytics

To address analytical needs, organizations often adopt BI solutions based on predefined dashboards and manually crafted queries. Figure 1 illustrates a representative architecture commonly found in such environments.

(1) Data exploration. End users formulate business questions — such as "What is the bug-resolution rate for a given project?", "How many commits were made in the last sprint?", or "How many user stories were added during a sprint?"—which must be translated by BI experts into analytical queries and data transformation using a traditional BI platform such as Pentaho.

(2) Dashboard delivery. The resulting datasets are presented through dashboards and reports used in day-to-day decision-making. Each dashboard component issues JSON queries to Pentaho, receiving curated data without exposing raw database complexity.

(3) Pentaho Semantic and processing layer. Behind these dashboards, Pentaho specialists maintain the semantic models, data pipelines, and API endpoints that map heterogeneous data sources into consistent analytical representations.

While this architecture enables basic reporting capabilities, it introduces some limitations. First, the process of creating or modifying indicators depends heavily on specialized expertise, making it slow and difficult to scale. Second, the reliance on predefined dashboards restricts exploratory analysis and limits flexibility. Third, maintaining and evolving the analytical infrastructure incurs significant operational costs.

These challenges are consistent with findings in the literature, which highlight barriers such as limited resources, data integration difficulties, and misalignment between analytical outputs and practitioners' needs [11]. In response, organizations increasingly explore automation and conversational interfaces to reduce manual effort and improve accessibility. However, designing solutions that are both flexible and reliable remains an open challenge.

4 Proposed Solution

The proposed approach aims to automate the translation of natural language business goals into CubeJS semantic models and queries, presenting results as text and visualizations. Section 4.1 explains the rationale behind the design choices, including the decisions to use CubeJS and Node-RED. Section 4.2 details the system architecture.

4.1 Design Rationale

The design of the proposed approach is motivated by the challenges illustrated in the motivating example (Section 3). Software analytics solutions must deliver timely insights without recurring reliance on BI experts, yet the legacy setup summarized in Figure 1 imposes a critical bottleneck: expert-driven semantic work. Answering new questions—such as "How many user stories were added during a sprint?" or "What is the average lead time for bug resolution?"—required a BI expert to inspect the schema, join relevant tables, and manually encode measures and dimensions, delaying decisions and discouraging experimentation.

CubeJS for semantic modeling. CubeJS offered a lightweight, developer-friendly semantic layer with on-premise deployment, real-time query support, and API integration. After connecting a database source, engineers annotate its tables as Cube elements—cubes, measures, dimensions, joins, and optional pre-aggregations—exposing higher-level analytical layers that let end users express business questions uniformly, with CubeJS translating them into efficient SQL-backed JSON queries without requiring custom ETL pipelines.

A cube represents a business entity or fact table and specifies (i) the physical SQL table; (ii) optional *joins* to related cubes; (iii) *measures* (numeric aggregations such as counts, sums, or averages); and (iv) *dimensions* for grouping, filtering, and drill-down. *Pre-aggregations* define materialized rollups that accelerate repeated queries. Listing 1 shows a simplified *orders* cube with joins to *users* and *products*, a drillable count measure, and `id`/`status` dimensions. When an external service queries "count orders grouped by `users.city`", CubeJS uses this definition to generate the necessary SQL, acting as a contract between raw relational data and higher-level analytics.

Early experiments. Initially, we envisioned a fully LLM-driven architecture using a retrieval-augmented generation (RAG) pipeline to translate natural-language questions directly into CubeJS JSON

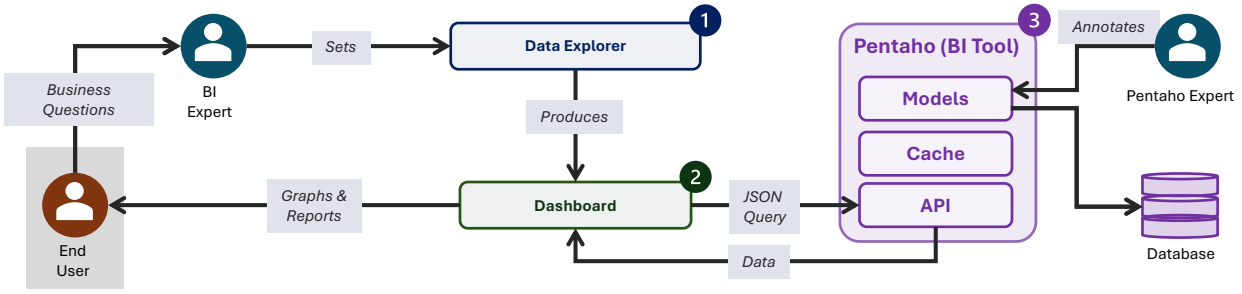

Figure 1: A Traditional Software Analytics Architecture using Pentaho (BI Tool).

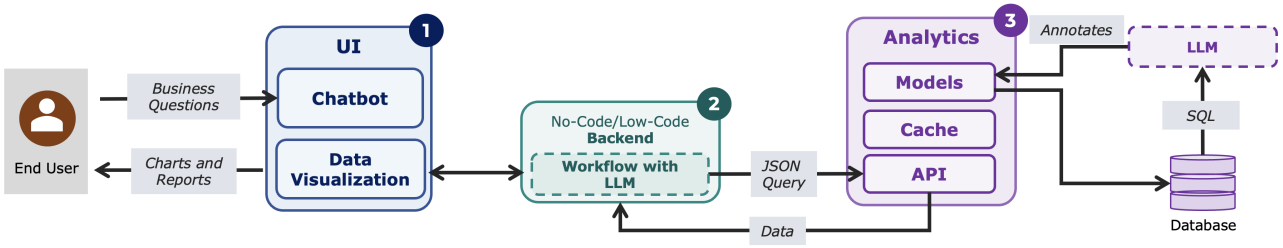

Figure 2: High-level architecture with LLMs, Node-RED, and CubeJS.

queries. While early tests showed promising accuracy, each analytic domain required its own curated retrieval pipeline—time-consuming and difficult to scale. Compact models such as LLaMA-3.1-8B also produced low-precision and inconsistent JSON outputs. These observations align with findings in the literature on LLM-based text-to-JSON generation: Escarda-Fernández et al. [9] report similar challenges in IoT monitoring, Chen et al. [7] document generation errors in extended-reality environments, and Neubauer et al. [19] address consistency issues through hybrid LLM and deterministic safeguards—reinforcing our decision to adopt a hybrid architecture.

Listing 1: CubeJS cube definition for the `orders` table

```
cubes:
  - name: orders
    sql_table: orders
    joins:
      - name: users
        relationship: many_to_one
        sql: "{CUBE}.user_id = {users.id}"
      - name: products
        relationship: many_to_one
        sql: "{CUBE}.product_id = {products.id}"
    measures:
      - name: count
        type: count
        drill_members:
          - id
          - status
          - products.name
          - users.city
    dimensions:
      - name: id
        sql: id
        type: number
        primary_key: true
      - name: status
        sql: status
        type: string
```

Hybrid NLP + LLM strategy. These limitations motivated a solution combining traditional NLP with targeted LLM support. We developed a SpaCy-based pipeline trained on 596 manually annotated questions in two stages: a property classifier identifying the target property, followed by an extractor capturing entities, time intervals, and measures. For the rewriting step, we adopted LLaMA-3.1-8B, a compact open-source model that, unlike large proprietary alternatives, can be self-hosted on modest hardware. In contrast to the earlier fully LLM-driven experiments, where JSON generation demanded high model capacity, question rewriting is a simpler task that compact models handle reliably, avoiding external API dependencies in the industrial deployment. The LLM is thus used solely to rewrite and simplify user questions before structured query generation. This hybrid design achieved robust accuracy with a simpler configuration and lower operational overhead.

Node-RED as orchestration backbone. Another decisive design choice was to adopt Node-RED as the engine for query translation and workflow management. Node-RED is an open-source, flow-based development tool widely used in IoT and industrial IoT (IIoT) contexts for connecting devices and services through protocols such as MQTT [5, 6, 10, 16, 21, 23, 24]. To the best of our knowledge, this approach has not been previously applied to conversational software analytics. As illustrated in the motivating example (Section 3), supporting flexible and evolving analytical queries requires an orchestration layer that is both modular and easy to adapt. In this context, Node-RED's visual, flow-based environment enables rapid creation and evolution of translation workflows, promotes component reuse, and facilitates collaboration among multidisciplinary teams, while reducing the need for custom code. By placing Node-RED at the core of the architecture, the approach provides a maintainable and extensible orchestration layer that cleanly integrates LLM-assisted semantic annotation with the SpaCy-based question parser.

The following subsections describe the overall architecture and each key component in detail.

4.2 Solution Architecture

Figure 2 summarizes our solution: a conversational analytics pipeline that turns natural-language questions into executable queries over

a semantic BI layer, returning both charts and textual explanations. The architecture comprises three layers: (1) User Interface (UI) *in blue*, (2) No-Code/Low-Code Backend *in green*, and (3) Analytics *in purple*.

4.2.1 User Interface Layer Practitioners type business questions in natural language; the UI forwards them as a single HTTP `POST` carrying the raw utterance. The response is a JSON object with two fields: `data` (tabular series rendered as tables or charts) and `info` (a Markdown narrative explaining the result), keeping the UI thin and decoupled from analytics logic. Figure 3 shows a real interaction: the user submits a question, the middle panel renders the `data` field as a chart and table, and a concise narrative from `info` appears as a "Result Highlight". The conversation metaphor lowers the barrier for non-BI users, the Markdown `info` serves as an auditable explanation, and client-side rendering lets visual components evolve independently of the pipeline.

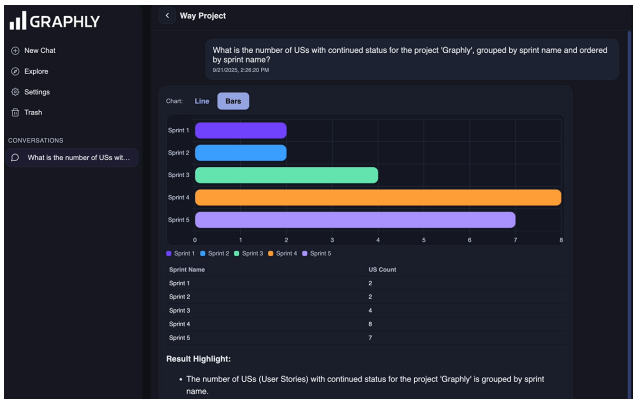

Figure 3: Chat-centric UI.

4.2.2 Low-Code/No-Code Backend Layer This layer (green in Figure 2) was implemented in Node-RED as a single observable pipeline (Figure 4). A request starts at *Listener* and ends at *Response* with a minimal JSON contract {`data`, `info`, `meta`}. The *Main Flow* routes the question through two subflows, normalizes time, executes the plan on CubeJS, and emits the response. The separation between *planning*, *execution*, and *narration* simplifies debugging and governance. The nodes composing the main flow (Figure 4) are described sequentially below.

- *Listener (I/O)*: receives the HTTP `POST`, attaches correlation id/auth/tenant, and rejects malformed payloads.
- *Cube Contextual History (Subflow 1)*: resolves analytic context over the CubeJS semantic layer.
- *Check Cache*: computes the embedding (All-MiniLM) and performs ANN lookup; on hit returns the stored CubeJS JSON; on miss forwards to synthesis.
- *Cube Query Builder (Subflow 2)*: synthesizes a fresh CubeJS JSON from the utterance.
- *Save Cache*: persists {embedding, normalized question, JSON} for paraphrases/near-duplicates.
- *Adjust Dates*: converts relative windows (e.g., "last sprint", "last 2 weeks") into explicit ISO ranges.
- *Request Executor*: invokes the CubeJS API with timeouts/retries and records execution telemetry.
- *Data Explainer*: computes basic statistics and generates the Markdown `info` with the LLM.
- *Response (I/O)*: returns the stable UI contract.
- *Error Monitor / Error Handler*: centralized path for retries, circuit breaking, and diagnostics.

Subflow **1) Cube Contextual History** loads CubeJS `/meta` via *Metadata Fetcher*, ranks candidate cubes against the question embedding (*Check Cube Context*), and carries short-term conversational memory for disambiguation (*Chat History*). Subflow **2) Cube Query Builder** prunes the schema with *Filter Cubes*, normalizes phrasing with the LLM *Rewriter*, and assembles the CubeJS JSON via the SpaCy-based *NL2CubeJS Tool*.

End-to-end, this layout mirrors the legend in Figure 4: *Generic* nodes (light blue) orchestrate control and I/O, *CubeJS-specific* nodes (purple) enforce the semantic contract, *Subflows* (salmon) encapsulate context and synthesis, *Error* nodes (red) route failures, and *I/O* nodes (yellow-green) mark boundaries. In practice, the semantic cache (*Check Cache*/*Save Cache*) short-circuits recomputation for paraphrased questions, yielding a **71.7%** reduction in end-to-end latency and an average response time of **15.36 s** in our deployment. The hybrid stack—comprising semantic matching (All-MiniLM), deterministic parsing/classification (SpaCy, with 596 labeled questions), and targeted LLM use (rewriting/explanation)—stabilizes JSON synthesis while keeping LLM calls focused. The low-code composition enables teams to evolve models or thresholds by swapping nodes without requiring refactoring of the pipeline.

4.2.3 Analytics Layer The analytics layer (purple in Figure 2) validates intent against a semantic model, executes queries over pre-aggregations, and returns tidy result sets. It exposes 15 business-facing cubes referencing a subset of 184 tables/views; several are decomposed into specialized sub-cubes so practitioners operate on fine-grained concepts (bugs, sprints, releases) rather than SQL.

- **Models**: CubeJS schemas defining *cubes*, *measures*, *dimensions*, and *joins*.
- **Cache (Pre-aggregations)**: server-side rollups accelerating repeated queries by sprint/project/time grain.
- **API**: stable JSON interface (`measures`, `dimensions`, `filters`, `timeDimensions`) with access control and execution metadata.
- **Database connection**: the only place where physical schemas are touched; computation is pushed to the warehouse when possible.

Planning and rewriting occur in the green box; validation, optimization, and governance reside in the purple box. Because semantics are centralized in *Models* and optimization in *Cache/Database*, the backend and UI can evolve independently without altering query logic or storage.

5 Study Setup and Environment

This section describes how the proposed approach was implemented and evaluated in an industrial setting. We present the development process (Section 5.1), the dataset (Section 5.2), and the evaluation measures (Section 5.3).

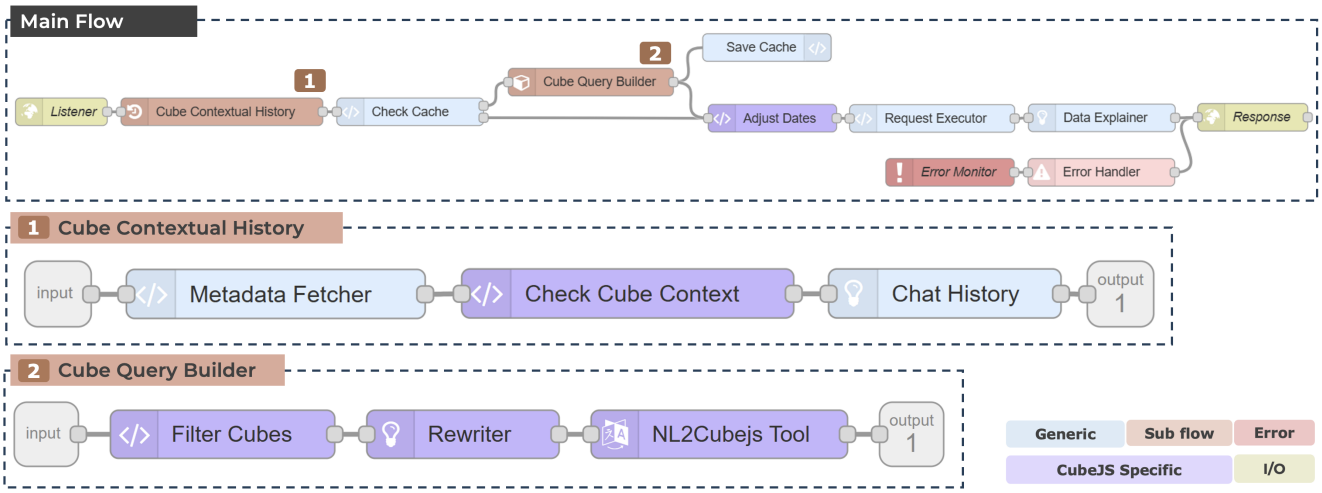

Figure 4: Main Flow of the No-Code/Low-Code Backend.

5.1 Development Process

We adopted an iterative, industry-in-the-loop process with weekly demos for project managers and data engineers from the studied industrial setting. The implementation follows the architecture in Figure 2 and is organized into four layers (Table 1). This layered design separates concerns—UI, orchestration/logic, analytics, and model serving, so that each can evolve independently and be deployed consistently via Docker across development, staging, and production.

The containerized architecture allows reproducible deployment across development, staging, and production. Basic automated checks ensure that new flows and configurations are syntactically valid and compatible with the existing CubeJS schema. Staging closely mirrors production and is used to validate new versions before deployment. This streamlined setup ensures that conversational analytics can be iteratively enhanced and safely rolled out while keeping the codebase maintainable and the operational footprint predictable.

5.2 Dataset

The evaluation used a production database from an industrial software project management platform, which consolidates operational artifacts of agile projects, such as requirements, issues, sprints, test executions, releases, and daily reports. The dataset contains data from 463 projects across multiple domains, including microservices, mobile, artificial intelligence, cloud, and IoT (see Table 2).

From the existing 184 tables and views capturing project-related activities, we focused on 15 that, based on interviews with domain experts, emerged as the primary sources of strategic and tactical questions. We mapped these into **15** CubeJS semantic cubes. The *bugs* domain is decomposed into three specialized cubes (resolution, status, and type) derived from the same underlying table. *Projects* and *phases* are modeled as two related cubes. *Tasks* include a dedicated *impediments* sub-cube. The remaining cubes cover *activities*, *dailies*, *issues*, *users*, *releases*, and *sprints*. Each cube was carefully annotated with measures, dimensions, joins, and pre-aggregations to expose the key business entities required to support management questions.

The test suite was constructed with 304 natural language questions, all in English, representing typical analytical tasks in the domains of software engineering and software security. The questions cover quantitative metrics, such as counts and sums (e.g., number of bugs closed or total resolution time) and derived statistics (e.g., mean and median cycle time), as well as distributions across categories (e.g., severity, bug type, assignee), temporal trends at different granularities (e.g., sprints, weeks, opening and closing dates), period-over-period or cross-project comparisons, filtering conditions (e.g., by project, priority, or team), and ordering criteria (e.g., from most to least critical, or from most recent to oldest). For each question, domain experts authored a reference CubeJS JSON plan and a short explanatory narrative. These artifacts serve respectively as the oracle for structural accuracy and as anchors for assessing explanation quality. To avoid leakage, the SpaCy classifier was trained on a separate, non-overlapping set of 596 labeled questions, and evaluation relied on a held-out split. Sensitive fields were anonymized, and filters containing personally identifiable information were excluded from caching.

5.3 Evaluation Measures

We evaluated the system with three sets of measures aligned to RQ1–RQ2 (query-generation accuracy, end-to-end latency, and cache impact), plus a Text-to-SQL baseline that contextualizes the pipeline's accuracy against a conventional direct-generation approach (Section 5.4). Unless stated otherwise, timings are wall-clock at the HTTP boundary (from *Listener* arrival to *Response* flush) and were collected in the staging topology described in Section 5.1. All experiments ran on a single on-premise server with an Intel(R) Xeon(R) Bronze 3204 CPU @ 1.90 GHz, 62 GB RAM, GPUs 6× Quadro RTX 4000 (8 GB) and 1× Quadro P2000 (5 GB), and storage 2× 896 GB SATA Intel SSDSC2KB96 SSDs.

Query-generation accuracy (RQ1). Each generated CubeJS JSON is executed against the database; the returned result set is

Table 1: Main technologies, responsibilities, and interfaces by system layer (colors correspond to Figure 2).

Layer (color)	Main technologies	Responsibilities / artifacts	Interfaces
UI (blue)	React, PrimeReact, Chart.js	Chat interface for natural-language questions; visualization of CubeJS results as tables and charts; English support	REST/JSON to backend
No-Code/Low-Code Backend (green)	Node-RED (flows), Python/Flask (helpers), spaCy (classifier), LLaMA-3.1-8B (question rewriting)	Orchestration of query translation; natural-language preprocessing and simplification; classification into CubeJS properties (measures, dimensions, filters, time/order); JSON assembly and execution	REST to CubeJS API; local HTTP to model service
Analytics (purple)	CubeJS (Models, Cache, API)	Semantic modeling over an industrial software project dataset (463 projects; 15 analytical cubes referencing a subset of 184 tables/views); pre-aggregations; access control; caching to reduce response times	CubeJS HTTP API; connectors to underlying relational database
Model serving	Ollama runtime hosting LLaMA-3.1-8B	Low-latency rewriting and question simplification in English	Local HTTP from backend flows
Infrastructure	Docker/Compose	Reproducible deployment across development, staging, and production; basic monitoring and logging	Container network

Table 2: Dataset overview and evaluation artifacts.

Component	Description	Value
Source schema	Industrial software project management platform database (issues, tests, releases, dailies, sprints, etc.)	184 tables/views; 463 projects
Semantic layer	CubeJS cubes covering the main analytical domains; some entities (e.g., bugs) are decomposed into multiple cubes	15 cubes
Question set	NL questions (English) with paraphrases	304
Task types	Count/sum; avg/median; distribution; trend; comparison	Balanced
Gold references	Expert-authored CubeJS JSON + narrative per question	1 per question
Training split	Separate SpaCy training set (non-overlapping)	596 questions
Privacy	Anonymized entities; PII-bearing filters not cached	Enforced

normalized (lexicographic row sorting, fixed numeric tolerance, ISO-8601 date normalization, explicit NULL handling) and compared for multiset equality against the reference result set produced by the expert-authored CubeJS JSON executed under identical conditions. A query counts as correct if and only if the normalized result sets match. This *execution accuracy* metric is the same protocol used for the Text-to-SQL baseline in Section 5.4, making the two systems directly commensurable. Confidence intervals are Wilson Score Intervals at the 95% level ($z = 1.96$). Per-property exact-match rates across CubeJS properties are reported in Table 3 as secondary diagnostic material.

End-to-end latency and phase breakdown (RQ2). We measure mean end-to-end latency and attribute time to instrumented phases that mirror the pipeline in Figure 4: *Cube Query Builder*, *Data Explainer*, *Cube Contextual History*, *Request Executor*, *Save Cache*, and *Check Cache*. For each phase, we report the mean time and its share of the total, summarized in Table 4. Unless explicitly noted (cache OFF experiments), latency numbers are collected with the semantic cache enabled.

Semantic-cache impact (RQ2). To isolate the cache effect, we run the same workload under *cache OFF* and *cache ON* regimes. We report the relative reduction in mean end-to-end latency when reusing validated plans. Between regimes we clear warm entries and randomize question order; prompts, dataset snapshot, and container graph remain fixed.

Data and prompts are fixed per release; synonym lists and similarity thresholds are versioned with the flows. Personally identifiable filters are never cached, and sensitive values are hashed in logs. All prompts were in English (EN); no bilingual runs were conducted.

5.4 Baseline: Text-to-SQL Evaluation

To provide a reference point for the proposed hybrid pipeline, we introduce a baseline that frames the same analytical task as a conventional Text-to-SQL problem. Instead of routing questions through the CubeJS semantic layer, the baseline hands each natural-language question directly to a language model and asks it to produce an executable MySQL query. This setup isolates the contribution of the semantic layer and the deterministic parsing components by removing them entirely from the translation path.

Table 3: Diagnostic per-property exact-match rates across CubeJS properties (95% Wilson CI).

CubeJS property	Accuracy (%)	95% CI
Measures	87	[82.8, 90.3]
Orders	81	[76.2, 85.0]
Time dimensions	78	[73.0, 82.3]
Dimensions	73	[67.8, 77.7]
Primary metric: execution accuracy 73.4% (95% CI [68.1%, 78.0%])		

The baseline was evaluated on the same 304 questions used for the hybrid pipeline, covering the five task types equally (count/sum, avg/median, distribution, trend, comparison), ensuring both systems are compared on identical inputs and expert-defined references. Since the expert references were authored in CubeJS, we converted them automatically to MySQL before evaluation. Each cube was mapped to its corresponding physical database table, filter conditions were translated to SQL predicates, and relationships between cubes became JOIN clauses. This conversion ensured that the baseline could be evaluated under the same execution accuracy protocol described in Section 5.3, making the results directly comparable to those of the hybrid pipeline.

The model chosen for the baseline is defog-llama3-sqlcoder-8b [8], a LLaMA 3 8B variant fine-tuned specifically for SQL generation. This choice rests on three grounds. First, the model is purpose-built for Text-to-SQL tasks and has been adopted as an experimental subject in recent work on LLM-based query generation [17, 30], confirming its standing as a representative of direct LLM-to-SQL approaches. Second, sharing the same model family and parameter count as the LLaMA-3.1-8B used in the hybrid pipeline ensures that accuracy differences reflect architectural choices rather than differences in model capacity. Third, the comparison isolates the architectural contrast: an end-to-end LLM-driven translation versus a hybrid pipeline that delegates generation to deterministic components grounded in a semantic layer. Every prompt supplies the

complete DDL of the underlying physical tables referenced by the 15 analytical cubes in the standard defog-sqlcoder format, giving the model the same structural information that the hybrid pipeline obtains from the CubeJS metadata endpoint.

The shared evaluation criterion for both the hybrid pipeline and this baseline is *Execution Accuracy*, defined in Section 5.3 and applied here under identical result-set normalization. This criterion captures semantic equivalences that a purely syntactic comparison would miss, since many distinct query strings can encode the same analytical intent.

6 Evaluation Results

This section presents the results for our RQs. We first assess *translation reliability* (RQ1) via execution accuracy of the system-generated CubeJS JSON against expert references (Section 5.1), with a diagnostic per-property breakdown provided as secondary material. Next, we analyze *responsiveness* (RQ2) via an end-to-end latency breakdown and a cache OFF/ON contrast under the staging setup described earlier (Section 5.2). Finally, we compare the hybrid pipeline against a direct Text-to-SQL baseline to contextualize both accuracy and latency (Section 5.3).

6.1 Translation Reliability (RQ1)

The hybrid pipeline achieves **73.4% execution accuracy** on the 304-question evaluation set (95% Wilson CI [68.1%, 78.0%]), evaluated under the same protocol used for the Text-to-SQL baseline (Section 5.4). Table 3 provides a secondary diagnostic breakdown of per-property exact-match rates between system-produced CubeJS JSON and expert references, together with 95% Wilson confidence intervals. Property-level accuracy is higher for *measures* (87%, CI [82.8, 90.3]) and *orders* (81%, CI [76.2, 85.0]), while *timeDimensions* (78%, CI [73.0, 82.3]) and *dimensions* (73%, CI [67.8, 77.7]) lag behind.

High agreement on *measures* indicates that the combination of semantic matching (All-MiniLM) with deterministic classification (spaCy) reliably maps user intent to measure definitions (e.g., `bugs.count`, `tasks.avgCycleTime`). Likewise, *orders* benefit from explicit lexical cues (e.g., "top", "ascending/descending", "latest"), which the classifier captures consistently.

Analysis of execution failures. The approximately 26.6% of queries that failed execution fall into four categories identified by manual inspection of the underlying CubeJS JSON: (E1) correct window but mismatched *granularity* (e.g., `day` vs. `week`), producing a result set with a different structure; (E2) correct dimension with incomplete *drill members*, causing extra or missing columns; (E3) missing disambiguating filter for project/team, returning a superset of the intended rows; and (E4) synonym resolved to a correlated field (e.g., `projects.code` instead of `projects.name`), yielding a different result set despite similar intent. E1–E2 dominate trend/-comparison queries; E3–E4 arise when organization-specific aliases were unseen during training.

Per-property diagnostics. The per-property breakdown in Table 3 is consistent with this query-level analysis: the weaker exact-match rates on *timeDimensions* and *dimensions* reflect the same underlying phenomena captured by categories E1 (temporal granularity) and E4 (long-tail synonyms), respectively.

Ablations and robustness. Internal tests (not shown) suggest that removing the *rewriter* (LLaMA-3.1-8B) increases E1/E3, while enlarging the synonym dictionary reduces E4. The semantic cache does not affect accuracy (it reuses validated plans) but only latency (addressed in RQ2).

These patterns support a semantics-first design for measure selection and ordering, while highlighting temporal normalization and organizational synonym coverage as primary areas for improvement. Two planned extensions are: (i) incorporating a project/sprint calendar into *Adjust Dates* to resolve relative anchors, and (ii) adding a lightweight entity linker backed by tenant-versioned dictionaries.

6.1.1 *Baseline Comparison: Text-to-SQL*

The hybrid pipeline is contrasted with the direct Text-to-SQL baseline described in Section 5.4, spanning accuracy, error profiles, and latency, as summarized in Figure 5.

Evaluated on the same 304-question set used for the hybrid pipeline, the baseline achieved an Execution Accuracy of 62.08% (95% CI [56.5%, 67.4%]). Mean per-query latency was 5.19 s end-to-end, dominated by model inference (range 4.77–9.25 s); MySQL execution contributed a negligible 2 ms.

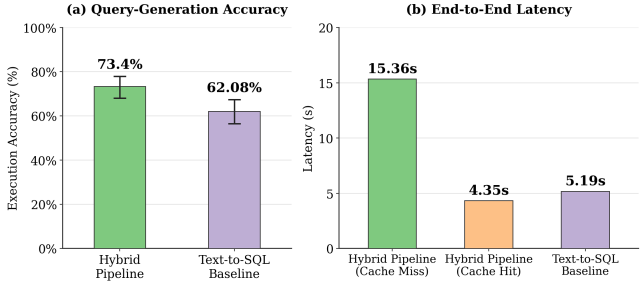

Figure 5: Comparison between the hybrid pipeline and the Text-to-SQL baseline. Panel (a) shows execution accuracy with 95% Wilson confidence intervals: hybrid pipeline 73.4% (CI [68.1%, 78.0%]), baseline 62.08% (CI [56.5%, 67.4%]); both are execution accuracy under the same result-set normalization protocol; intervals do not overlap. Panel (b) reports end-to-end latency for the hybrid pipeline under cache-miss and cache-hit conditions alongside the baseline mean latency.

Both the hybrid pipeline and the Text-to-SQL baseline were evaluated using execution accuracy on the identical 304-question set, against the same expert-authored references, and under the same result-set normalization protocol defined in Section 5.3. Figure 5 (panel a) makes the accuracy gap visually clear: the hybrid pipeline achieves 73.4% execution accuracy (95% Wilson CI [68.1%, 78.0%]) while the baseline achieves 62.08% (95% Wilson CI [56.5%, 67.4%]). The gap of approximately 11.3 percentage points is statistically significant, as the two Wilson 95% CIs do not overlap. Because both systems are assessed under the same metric, this difference directly reflects the architectural advantage of grounding query generation in a semantic analytics layer.

The latency comparison, shown in Figure 5 (panel b), reveals a trade-off. On a cache miss, the hybrid pipeline's mean latency of 15.36 s exceeds the baseline's mean latency of 5.19 s, reflecting the additional cost of context resolution, question rewriting, classification, and narrative generation. However, the semantic cache reduces latency for repeated or paraphrased queries by 71.7%, bringing the effective response time to approximately 4.35 s—comparable to the baseline. Moreover, every hybrid-pipeline response includes

an explanatory narrative that contextualizes the numerical result, a feature the SQL baseline does not offer.

From an architectural perspective, the baseline concentrates translation entirely inside the language model, while the hybrid pipeline distributes this burden across deterministic parsing (spaCy), semantic matching (All-MiniLM), constrained LLM rewriting, and schema-level validation (CubeJS), trading pipeline complexity for higher reliability.

Answer to RQ1 — Translation Reliability

The hybrid pipeline (All-MiniLM for semantic matching, spaCy for parsing/classification, LLaMA-3.1-8B for rewriting) achieves **73.4% execution accuracy** (95% Wilson CI [68.1%, 78.0%]) on 304 questions. Residual errors (approximately 26.6%) stem mainly from relative time spans and long-tail dimension synonyms; schema validity is enforced by CubeJS. The defog-llama3-sqlcoder-8b Text-to-SQL baseline, evaluated under the same execution-accuracy protocol on the identical 304-question set, achieves 62.08% (95% Wilson CI [56.5%, 67.4%]). The non-overlapping confidence intervals (Figure 5) confirm, under a unified metric, the benefit of grounding query generation in a semantic analytics layer.

6.2 Performance and Cache Impact (RQ2)

Table 4 decomposes end-to-end wall-clock latency by instrumented phase. The critical path is dominated by the *Cube Query Builder* (planning and JSON synthesis; 71.7% of total) and the *Data Explainer* (LLM narrative; 17.8%). In contrast, the *Request Executor* contributes under 1%, indicating that CubeJS pre-aggregations and rollups keep execution time low once a valid plan is established. Two regimes explain the observed behavior. On *cache misses*, the pipeline pays the full planning cost (embedding, rewriting, classification, JSON assembly), which drives the 11.01 s share for the builder. On *cache hits*, the validated plan is reused and the *Cube Query Builder* (11.01 s, 71.7% of total) is bypassed; only the remaining blocks (*Cube Contextual History*, *Request Executor*, *Data Explainer*, and *Save Cache*) execute, totalling approximately 4.35 s and yielding a 71.7% latency reduction relative to the cache-miss baseline. Practically, responsiveness is bounded mainly by (i) plan synthesis when no reusable plan exists and (ii) LLM explanation time.

Table 4: Average execution time per processing block.

Block / Stage	Avg. Time (s)	Share of Total (%)
Cube Query Builder	**11.01**	**71.7**
Data Explainer	2.73	17.8
Cube Contextual History	1.43	9.3
Request Executor	0.14	0.9
Save Cache	0.03	0.2
Check Cache	0.02	0.1
Total	**15.36**	**100**

Implications. Optimizing for interactive use should prioritize: (P1) reducing cold-path planning time (e.g., lighter/quantized rewriter, faster classifiers, batched embeddings); (P2) increasing hit rate via stricter canonicalization and tuned similarity thresholds; and (P3) constraining explanation length or adopting a faster summarizer for small result sets. Since execution is already inexpensive, further database-level tuning yields diminishing returns compared to planning and narration.

Practical takeaways. Making the system feel "instant" hinges more on (i) reusing validated plans and (ii) trimming narration cost than on database tuning. Concretely, we recommend: caching at the level of *normalized question→JSON plan*; lowering LLM temperature and max-tokens for explanations; and adopting lightweight rerankers to raise near-duplicate detection. These interventions target the two dominant phases in Table 4 and are orthogonal to warehouse scale, which the current pre-aggregation strategy already shields.

Answer to RQ2 - Performance and Cache Impact

Across 15 analytical cubes referencing a subset of 184 tables/views, mean end-to-end latency under cache-miss conditions is **15.36 s**. Reusing semantically equivalent plans via the cache reduces latency for repeated/paraphrased queries by **71.7%**. CubeJS rollups keep execution time negligible; the main optimization target is planning (JSON synthesis) and, secondarily, LLM narration. Qualitative observations on question complexity and explanation length suggest prioritizing plan canonicalization and a tiered summarization policy for sustained interactivity.

7 Implications for Research and Practice

In this section, we discuss some implications of our results for the research and practice on the topic of conversational software analytics.

Implications for Research. Our results show that hybrid architectures combining deterministic NLP with targeted LLM support achieve higher reliability than fully LLM-driven approaches, reaching 73.4% execution accuracy and outperforming a Text-to-SQL baseline. This reinforces the need to constrain LLM usage in structured tasks.

We also highlight the importance of semantic layers (e.g., CubeJS) to ground query generation, improving schema alignment and enabling validation. This suggests prioritizing schema-aware, multi-stage designs over end-to-end generation.

Finally, our findings indicate that low-code orchestration (e.g., Node-RED) supports modular and maintainable pipelines, while semantic modeling remains a key open challenge, still requiring expert knowledge.

Implications for Practice. The results indicate that conversational analytics can reduce dependence on BI experts and accelerate access to insights, enabling tasks that previously took days to be completed in hours.

Low-code orchestration provides a practical way to build and evolve such pipelines, reducing adoption barriers. Additionally, performance results show that responsiveness depends primarily on plan reuse: semantic caching reduced latency by 71.7%, suggesting that practitioners should prioritize caching and orchestration over database tuning.

Despite these benefits, domain expertise remains essential for defining semantic models and ensuring governance, requiring collaborative workflows between LLMs and experts.

In summary, combining semantic layers, hybrid NLP techniques, and low-code orchestration enables scalable conversational analytics, while highlighting open challenges in semantic modeling.

8 Threats to Validity

We analyze threats following standard categories: construct, internal, external, and conclusion validity [26].

Construct validity. Our accuracy measure (execution accuracy) counts a query as correct if and only if the normalized result

sets match. Different CubeJS queries may produce identical result sets despite structural differences; this intentionally credits functional equivalents and is applied symmetrically to both the hybrid pipeline and the Text-to-SQL baseline. To verify that execution failures reflect true semantic errors, we manually inspected a sample of mismatches and found that most corresponded to genuine semantic errors rather than normalization artifacts. Future work could complement execution accuracy with semantic-equivalence checking for a finer-grained analysis.

Internal validity. The evaluation depends on a manually defined set of CubeJS cubes and a question suite authored by domain experts. Modeling mistakes in the cubes or inadvertent overlap between training and evaluation questions could bias results. We minimized this risk by double-checking all cubes and using a held-out test set, but residual bias cannot be ruled out. Replication requires access to the same cube definitions and test questions; to foster reproducibility, all evaluation artifacts are version-controlled and can be shared under appropriate agreements.

External validity. This study was conducted in a single industrial setting, using a dataset of 463 software projects modeled through 15 analytical cubes built over a larger database schema. Although the dataset spans multiple domains (e.g., web, mobile, AI, and IoT systems), the results may not generalize to organizations with different data structures, development processes, or governance practices. Furthermore, all evaluation queries were written in English, and the system was not assessed under multilingual conditions. Thus, generalizing the approach to additional languages and substantially different data models may require adaptation. Consequently, while the architectural insights are broadly relevant, the reported accuracy and latency should not be generalized without caution.

Conclusion validity. Performance and latency figures (e.g., 73.4% execution accuracy and 71.7% latency reduction from semantic caching) were obtained in a controlled environment with specific hardware and CubeJS configurations. Different deployments may yield different absolute values. Replicating these measurements under diverse operational loads would strengthen the statistical confidence in our conclusions. Future studies should replicate the experiments under diverse conditions, incorporate statistical analysis, and compare against alternative query-generation methods to strengthen confidence in the findings.

9 Final Remarks

This paper presented an empirical study of a conversational analytics approach for software project management to investigate whether non-BI stakeholders can effectively formulate natural-language queries and obtain both executable queries and narrative explanations without relying on specialist-maintained dashboards. Our approach combines a semantic BI layer (CubeJS), a low-code orchestration backend (Node-RED), and a hybrid NLP stack integrating semantic similarity matching (All-MiniLM), rule-based parsing and classification (spaCy), and targeted LLM usage (LLaMA-3.1-8B) for question rewriting and result explanation. The approach was evaluated in an industrial setting using real project data and expert-defined reference queries.

The approach achieves reliable query translation and interactive responsiveness, with 73.4% execution accuracy (95% Wilson CI [68.1%, 78.0%]) on 304 questions. Per-property results (Table 3) show the highest exact-match rates for *measures* and *orders*, and lower agreement for *timeDimensions* and *dimensions*. A controlled comparison with a Text-to-SQL baseline fine-tuned specifically for SQL generation yielded 62.08% execution accuracy (95% Wilson CI [56.5%, 67.4%]) on the identical 304-question set, evaluated under the same protocol. The non-overlapping confidence intervals indicate that grounding query generation in a semantic analytics layer improves translation reliability. End-to-end mean latency under cache-miss conditions is 15.36 s on a single on-prem server across 15 analytical cubes. A semantic cache that reuses validated JSON plans reduces latency for repeated or paraphrased queries by 71.7%, enabling more responsive interactions.

From a research perspective, the findings support a *hybrid* design pattern for text-to-analytics, in which semantics and parsing are kept deterministic whenever possible, LLMs are restricted to normalization and explanation tasks, and both are integrated through a semantic layer that enforces schema validity. From a practical perspective, the results suggest that low-code orchestration (Node-RED) is effective for building, observing, and evolving multi-step conversational pipelines with modest effort. The main performance lever is not database tuning but (i) plan reuse via semantic caching and (ii) trimming narration cost. Finally, while LLMs accelerate authoring and maintenance, *conceptual modeling* of metrics and joins remains an expert task and a key governance boundary.

For future work, we plan to: (1) expand coverage to additional analytical cubes and indicators; (2) incorporate calendar-aware temporal normalization and lightweight entity linking to reduce errors in *timeDimensions* and long-tail *dimensions*; (3) explore more efficient LLM configurations, including quantized rewriters and tiered explanation policies, to reduce planning and narration latency; (4) conduct broader user studies, A/B evaluations with larger and multilingual workloads, and comparisons against additional Text-to-SQL baselines and commercial LLMs to further validate the hybrid architecture; (5) extend support to additional languages beyond English, evaluating cross-lingual generalization and adaptation requirements; and (6) investigate semi-automatic support for semantic model evolution (e.g., suggested measures and joins, safety checks) to further reduce expert effort while preserving correctness and governance.

Artifact Availability

The dataset and supplementary materials are available in an anonymized repository: https://anonymous.4open.science/r/D9B2.

Acknowledgements

The authors acknowledge the use of GPT-based tools to support text anonymization, and to generate LaTeX code for “Listing 1” and the summary boxes (“Answer to RQ1” and “Answer to RQ2”).

This work has been partially funded by the project “*iSOP Base: Investigação e desenvolvimento de base arquitetural e tecnológica da Intelligent Sensing Operating Platform (iSOP)*” supported by CENTRO DE COMPETÊNCIA EMBRAPII VIRTUS EM HARDWARE INTELIGENTE PARA INDÚSTRIA - VIRTUS-CC, with financial resources from the PPI HardwareBR of the MCTI grant number 055/2023, signed with EMBRAPII.

References

[1] Tamer Mohamed Abdellatif, Luiz Fernando Capretz, and Danny Ho. 2015. Software Analytics to Software Practice: A Systematic Literature Review. In *2015*

IEEE/ACM 1st International Workshop on Big Data Software Engineering. 30–36. doi:10.1109/BIGDSE.2015.14

[2] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 201–210. doi:10.1145/3661167.3661218

[3] Sathurshan Arulmohan, Marie-Jean Meurs, and Sébastien Mosser. 2023. Extracting Domain Models from Textual Requirements in the Era of Large Language Models. In *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 580–587. doi:10.1109/MODELS-C59198.2023.00096

[4] Katarzyna Biesialska, Xavier Franch, and Victor Muntés-Mulero. 2021. Big Data analytics in Agile software development: A systematic mapping study. *Information and Software Technology* 132 (2021), 106448.

[5] Michael Blackstock and Rodger Lea. 2014. Toward a distributed data flow platform for the web of things (distributed node-red). In *Proceedings of the 5th International Workshop on Web of Things*. 34–39.

[6] Somphop Chanthakit and Choopan Rattanapoka. 2018. Mqtt based air quality monitoring system using node MCU and node-red. In *2018 Seventh ICT International Student Project Conference (ICT-ISPC)*. IEEE, 1–5.

[7] Jiangong Chen, Xiaoyi Wu, Tian Lan, and Bin Li. 2025. LLMER: Crafting Interactive Extended Reality Worlds with JSON Data Generated by Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* 31, 5 (2025), 2715–2724. doi:10.1109/TVCG.2025.3549549

[8] Defog AI. 2024. Llama-3-SQLCoder-8B: A Fine-tuned Model for Text-to-SQL Generation. https://huggingface.co/defog/llama-3-sqlcoder-8b. Accessed: 2024.

[9] Miguel Escarda-Fernández, Iñigo López-Riobóo-Botana, Santiago Barro-Tojeiro, Lara Padrón-Cousillas, Sonia Gonzalez-Vázquez, Antonio Carreiro-Alonso, and Pablo Gómez-Area. 2024. Llms on the fly: Text-to-json for custom api calling. *Proceedings of the SEPLN-CEDI* (2024).

[10] Katalin Ferencz and József Domokos. 2019. Using Node-RED platform in an industrial environment. *XXXV. Jubileumi Kandó Konferencia, Budapest* (2019), 52–63.

[11] Iris Figalist, Christoph Elsner, Jan Bosch, and Helena Holmström Olsson. 2022. Breaking the vicious circle: A case study on why AI for software analytics and business intelligence does not take off in practice. *Journal of Systems and Software* 184 (2022), 111135. doi:10.1016/j.jss.2021.111135

[12] Wei Fu, Tim Menzies, and Xipeng Shen. 2016. Tuning for software analytics: Is it really necessary? *Information and Software Technology* 76 (2016), 135–146. doi:10.1016/j.infsof.2016.04.017

[13] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 124 (May 2025), 30 pages. doi:10.1145/3712003

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[15] Arsham Gholamzadeh Khoee, Shuai Wang, Yinan Yu, Robert Feldt, and Dhasarathy Parthasarathy. 2025. Gatelens: A reasoning-enhanced llm agent for automotive software release analytics. *arXiv preprint arXiv:2503.21735* (2025).

[16] Milica Lekić and Gordana Gardašević. 2018. IoT sensor integration to Node-RED platform. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 1–5. doi:10.1109/INFOTEH.2018.8345544

[17] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering* (2025).

[18] Silverio Martínez-Fernández, Anna Maria Vollmer, Andreas Jedlitschka, Xavier Franch, Lidia López, Prabhat Ram, Pilar Rodríguez, Sanja Aaramaa, Alessandra Bagnato, Michał Choraś, and Jari Partanen. 2019. Continuously Assessing and Improving Software Quality With Software Analytics Tools: A Case Study. *IEEE Access* 7 (2019), 68219–68239. doi:10.1109/ACCESS.2019.2917403

[19] Felix Neubauer, Jürgen Pleiss, and Benjamin Uekermann. 2025. AI-assisted JSON Schema Creation and Mapping. *arXiv preprint arXiv:2508.05192* (2025).

[20] Mirko Perkusich, Lenardo Chaves e Silva, Alexandre Costa, Felipe Ramos, Renata Saraiva, Arthur Freire, Ednaldo Dilorenzo, Emanuel Dantas, Danilo Santos, Kyller Gorgônio, et al. 2020. Intelligent software engineering in the context of agile software development: A systematic literature review. *Information and Software Technology* 119 (2020), 106241.

[21] Anoja Rajalakshmi and Hamid Shahnasser. 2017. Internet of Things using Node-Red and alexa. In *2017 17th International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, 1–4.

[22] Thiago Rique, Mirko Perkusich, Emanuel Dantas, Danyllo Albuquerque, Kyller Gorgônio, Hyggo Almeida, and Angelo Perkusich. 2023. On Adopting Software Analytics for Managerial Decision-Making: A Practitioner's Perspective. *IEEE Access* 11 (2023), 73145–73163. doi:10.1109/ACCESS.2023.3294823

[23] Sabrina Sicari, Alessandra Rizzardi, and Alberto Coen-Porisini. 2019. Smart transport and logistics: A Node-RED implementation. *Internet Technology Letters* 2, 2 (2019), e88.

[24] Mohamed Tabaa, Brahim Chouri, Safa Saadaoui, and Karim Alami. 2018. Industrial communication based on modbus and node-RED. *Procedia computer science* 130 (2018), 583–588.

[25] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 571–583. doi:10.1145/3643991.3644918

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[27] Chaoyun Zhang, Zicheng Ma, Yuhao Wu, Shilin He, Si Qin, Minghua Ma, Xiaoting Qin, Yu Kang, Yuyi Liang, Xiaoyu Gou, Yajie Xue, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. AllHands :Ask Me Anything on Large-scale Verbatim Feedback via Large Language Models. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 43–57. doi:10.1109/ICDE65448.2025.00011

[28] Dongmei Zhang, Yingnong Dang, Jian-Guang Lou, Shi Han, Haidong Zhang, and Tao Xie. 2011. Software analytics as a learning case in practice: approaches and experiences. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (Lawrence, Kansas, USA) *(MALETS '11)*. Association for Computing Machinery, New York, NY, USA, 55–58. doi:10.1145/2070821.2070829

[29] Dongmei Zhang, Shi Han, Yingnong Dang, Jian-Guang Lou, Haidong Zhang, and Tao Xie. 2013. Software Analytics in Practice. *IEEE Software* 30, 5 (2013), 30–37. doi:10.1109/MS.2013.94

[30] Lu Zhao, Xin Jiang, Feng Tao, Wen Liu, Xue Wang, and Yu Hao Wu. 2024. Research on Large Model Text-to-SQL Optimization Method for Intelligent Interaction in the Field of Construction Safety. *IEEE Access* (2024). doi:10.1109/ACCESS.2024.10810146

[31] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50.