ISSN: 2581-3404 (Online) IF: 5.86 (SJIF)
IJIRTM, Volume-10, Special Issue-2, April-2026

Impact of Database Management Systems on Industrial Performance

Kartik[1], Sushant[2], Shubham Panghal[3], B.K. Verma[4]

Research Scholar, Department of Computer Science & Engineering (AI & DS), Panipat Institute of Engineering and Technology, Panipat, India [1,2,3]

Head & Professor, Department of Computer Science & Engineering (AI & DS), Panipat Institute of Engineering and Technology, Panipat, India [4]

ikartikchhoker@gmail.com[1], sushantclg846@gmail.com[2], shubhampanghal.work@gmail.com[3]

Abstract. *The majority of manufacturing enterprises manage operational data in separate spreadsheets, paperwork, and standalone software. Such segregation leads to time-consuming processes, data duplication, and a continuous divide between factory floor activity and management oversight. This article outlines the development and deployment of FactoryFlow, a full-stack web app developed to analyze the impact of a specially-designed relational DBMS on organizational performance within the realm of manufacturing. The application employs Next.js 15, PostgreSQL 16 on Neon Cloud, and Prisma ORM to unify employee records, batch production, departmental structures, authorization, and accounting information in a normalized database. The FactoryFlow AI Command Center powered by Groq API (LLaMA 3.3 70B) incorporates four analytical modules, including predictive maintenance, resource optimization, quality control analysis, and performance benchmarks. The experiment revealed that core DBMS functions performed within less than 200 ms, while AI analysis returned within three seconds. The findings provide substantial evidence that the implementation of a DBMS, coupled with AI analytics, positively impacts data accessibility, decision-making pace, and operational visibility within industrial enterprises.*

Keywords: Database Management Systems, Industrial Performance, PostgreSQL, Prisma ORM, Groq API, Next.js, KPI Dashboard, Predictive Analytics, Production Tracking, Audit Logging.

Introduction

Most manufacturing firms have their operational data distributed among numerous spreadsheets, paper journals, and other separate software tools. While production data is stored in one document, employee details are recorded in another, and financial indicators in yet another document. This distribution leads to serious issues, including double entry of numbers, errors that can take several weeks to discover, and reports that have been compiled based on outdated information rather than actual data.

FactoryFlow was created specifically to solve these problems. The project is called "Impact of Database Management Systems on Industrial Performance" and seeks to explore precisely how a manufacturing

company can benefit from the introduction of a DBMS and abandoning their current data management methods. FactoryFlow is an attempt to prove this theoretical idea by creating an actual system. The heart of FactoryFlow is a web-based tool that combines all the company's information into a single PostgreSQL database running on Neon Cloud. We used Next.js 15 for the application's front-end and API layer since it allowed us to maintain simplicity without sacrificing performance. Rather than running a As for the code organization, the back-end server was separated from other codebases. Thus, all API routes and UI components resided in one repository, allowing us to develop the project much more quickly.

Another feature that makes our database management system better compared to other simple systems is the presence of AI functionality. By using the Groq API and the LLaMA 3.3 70B model, we were able to create a total of four analytical tools. Namely, they are predictive maintenance notifications, resource optimization recommendations, quality control, and performance benchmarking engine. The data needed by these tools is fetched in real time from PostgreSQL and structured recommendations are provided in just a few seconds. There is also a chatbot through which the manager can pose any question like 'Which shift had the highest defect rate this week', receiving a relevant response instead of empty cells. In the following sections of the research, we will describe the design decisions, system architecture, testing procedures, as well as its results and limitations.

2. Related Work

In addition to developing our solution, we looked into other references that influenced the way our data schema was developed and the technologies and AI components used in our implementation. The seven references are outlined below.

Elmasri and Navathe [1] was our main reference for developing the schema. Their description of the entity-relationship model and their explanation of the normalization process, especially the third normal form, helped us build the schema of the eight entities in FactoryFlow. At no point did we choose to prematurely denormalize; during our experiments, we had merged certain fields that would later need to be separated when writing complicated queries.

Transaction management theory presented by Silberschatz, Korth, and Sudarshan [2] was crucial when designing our transactions. What we learned from this book is the importance of putting multiple steps in a transaction instead of trying to execute everything one by one, which would lead to problems in case any operation failed. The documentation for web frameworks was critical for the development process. While the concept of App Router was novel to us at first, it proved quite helpful for organizing the codebase by having each route and its corresponding API handler in close proximity, thus minimizing the boilerplate code required. The Prisma ORM documentation [4] was vital in recognizing the benefits of schema migrations. At an early stage, we had to modify the Production table layout twice due to our ongoing refinement of the KPIs calculation criteria. Without schema migrations, the process of doing this would be considerably more difficult. The Neon Cloud documentation regarding PostgreSQL [5] was useful in introducing serverless database deployment. The biggest advantage of this approach we noted was how the database scaled up automatically during load testing.

In particular, the documentation on the Groq API [6] helped us in developing the AI Command Center significantly. Indeed, Groq inference works really quickly, and our tests show that most answers were returned in no more than two seconds, even when the prompt included information from 60 days of production history. Lastly, we looked into contemporary publications concerning the implementation of

AI-based DBMS in industry [7]. One notable conclusion to be drawn from several scientific sources is that efficiency increased by 15-30% when AI solutions were used alongside structured data systems. Although it cannot be taken as a definite number, this served as an additional reason for using AI in Factory Flow.

Table 1: Literature Survey Summary

Ref	Authors / Title	Year & Method	Finding & Relevance
[1]	Elmasri & Navathe – Fundamentals of Database Systems, 7th Ed.	2022 | Relational Model & Normalization	Core relational theory and 3NF normalization. FactoryFlow's schema for Employee, Department, Role, and Production entities is designed following these principles to eliminate data redundancy.
[2]	Silberschatz, Korth & Sudarshan – Database System Concepts, 7th Ed.	2020 | Transactions & Query Optimization	Transaction management and query optimization. FactoryFlow uses Prisma ORM atomic transactions for production batch inserts and indexed PostgreSQL queries for sub-100 ms KPI retrieval.
[3]	Next.js Documentation – App Router, API Routes, Server Components	2024 | Full-Stack Web Framework	Next.js 15 App Router enables SSR and co-located REST API routes. FactoryFlow leverages this for fast dashboard loads and all DBMS operations without a separate backend server.
[4]	Prisma ORM Documentation – Schema, Migrations, Client	2024 | Type-Safe ORM	Prisma provides type-safe DB access and versioned schema migrations. FactoryFlow uses Prisma Client across all 8 entities, preventing runtime type errors in production data operations.
[5]	PostgreSQL 16 / Neon Documentation – Serverless Postgres	2024 | Serverless Cloud Database	Neon's serverless PostgreSQL enables auto-scaling and connection pooling. FactoryFlow's database scales on demand during peak production batch ingestion without manual infrastructure management.
[6]	Groq AI API Documentation – LLaMA 3.3 70B Streaming	2024 | LLM API Integration	Groq's ultra-low latency LLM inference enables real-time AI analytics. FactoryFlow's AI Command Center uses streaming JSON for predictive maintenance and anomaly detection outputs.
[7]	Research Papers on AI-Driven Industrial DBMS and Predictive Analytics in Manufacturing	2023 | AI + DBMS Survey	Literature confirms 15–30% efficiency gains from AI-augmented DBMS in manufacturing. Validates FactoryFlow's architecture combining PostgreSQL with an LLM analytics intelligence layer.

3. System Architecture

A. Overall Design

The app is a full-stack web application where both the front-end and API layer are deployed using Vercel services, and the database is hosted on Neon's serverless PostgreSQL service. Communication between the database and front-end occurs through HTTP requests, where the React front-end sends request to Next.js API routes and fetches the data from the database using Prisma ORM. Thus, FactoryFlow does not have a separate backend server. The combination of these three services made deployment relatively straightforward for us.

The database consists of eight tables, namely Employee, Department, Role, Product, Production, PromotionHistory, AuditLog, and Dataset. Almost all important data gets stored in the Production table since all batch entries that are logged get stored there, and most of the KPIs shown in the dashboard come from the Production table. The AuditLog table logs all writing operations performed throughout the application. Above the database is the AI Command Center. It's just four report generators that get some information from the PostgreSQL database, format a prompt, and then send it to Groq's LLaMA 3.3 70B language model. The output comes as a JSON response, which we then parse for display in the frontend. Another component we have is a floating chat widget that sends the user input and any necessary data from the database in a prompt to the same model.

B. Frontend

For the frontend, we have React version 19 along with Next.js version 15 using its App Router. When loading for the first time, pages use server rendering to ensure fast performance even when connecting on slow networks. There is client-side navigation throughout the application between its six sections: Dashboard, AI Command Center, Employees, Production, Departments/Roles, and Reports.

Styling was entirely managed via Tailwind CSS. CSS variables were employed to implement light and dark themes, while Framer Motion was incorporated to add some transition animations to improve the streaming AI responses. As for graphs, the solution utilized Recharts library along with custom-made gauges using SVG for displaying efficiency and KPI indicators on the dashboard.

C. Backend & Database Access

All requests to APIs pass through Next.js route handlers that run in the serverless Node.js environment on Vercel. All route handlers are split into several modules by the domain they manage – one per each group of requests (employee handling, production processing, AI engine operation, report generation). The latter use Prisma Client to access data from PostgreSQL database. Being automatically generated from the schema by Prisma, the types helped us to detect quite a number of type mismatching errors early on. The approach used by Prisma, i.e., the soft delete mechanism (by recording deletedAt time), was quite helpful for the employee table. Data related to employees cannot just be deleted from the system because you need to retrieve the history of the employees in the company. We used this approach for some other tables too that needed their history to be maintained.

Database

Production table is the core part of the system. In the production table, we have the number of units produced, the number of defects, date of the batch and Product Id. Using all these values, API calculates efficiency score, total amount earned by the company and other metrics such as Cost per Unit, Margin etc which are not stored in the database at all.

The Audit Log tracks all INSERTS, UPDATES, and DELETES, complete with time stamp and the ID of the person executing the query. In actuality, this turned out to be one of the most helpful tools during our test period, since when there was an anomaly in the dashboard display, we could always find out what caused it.

E. Authentication and Security

We implemented RBAC ourselves rather than through some external auth provider. Executive-level users have access to all pages, as well as to the AI command center. User-level operators are restricted to recording batch production processes as well as viewing their own log entries. Before running any queries on the database, middleware will verify the user's role on each call to the API. This was easy to implement, but needs to be beefed up for production use.

4. Implementation

A. Technology Stack

- The tech stack decisions below were made based primarily on familiarity, documentation availability, and maintaining simplicity for our deployment:
- Next.js 15 + React 19: manages both the front end and the API layer. The use of the new App Router allowed us to do away with the Express server entirely.
- Postgres 16 + Neon Cloud: the serverless deployment made it unnecessary for us to manage the database server. Connection pooling was set up in the Neon Cloud dashboard.
- Prisma ORM 6: helped us achieve type-safe queries and an efficient migration process. Making schema adjustments was far easier using Prisma than it would have been via raw SQL.Groq API with LLaMA 3.3 70B: chosen for inference speed. Alternatives we tested were noticeably slower when handling large prompt payloads.
- Tailwind CSS plus Framer Motion: utility classes kept the CSS organized, and Framer handled the few animations we needed.
- Recharts plus custom SVG: Recharts covered standard line and bar charts; we wrote custom SVG for the circular efficiency gauge because no library produced exactly what we wanted.
- Vercel: zero-config deployment from GitHub. Every push to main triggered a fresh deploy automatically.

Table 2: Software Technology Stack

Category	Technology
Framework	Next.js 15 (React 19, App Router)
Database	PostgreSQL 16 — Neon Cloud (serverless)
ORM	Prisma ORM 6
AI / LLM	Groq API — LLaMA 3.3 70B Versatile (streaming)
Styling	Tailwind CSS, Framer Motion, CSS Variables
Charts & Visualization	Recharts, custom SVG gauges

Auth / Security	Role-based access control, audit logging, Prisma soft-delete
Dev Tools	VS Code, Git, GitHub, Postman
Deployment	Vercel (frontend + API) + Neon Cloud (database)
OS	Windows 11 / Linux (Ubuntu)

B. API Integrations

There is also an API module for each of them. For instance, the Employee module involves creating, updating, and even deleting employees' records. It is also responsible for making logs about the promotion history in the PromotionHistory table. Any update or deletion occurs within the Prisma transaction during which the AuditLog entity is created.

The Production module involves receiving information about products such as ID, production volume, defective products, and date. Afterward, the server makes all computations, and the frontend does not have to perform any calculations and simply shows what the API returns. Of all the modules, the AI Command Center module is the most complex. We first load the respective records for each analytical engine from the PostgreSQL database, convert them to JSON format, enclose them within a prompt asking Groq for the particular type of analysis needed, and send the request using stream support. Once we receive the tokens from Groq, the front end renders the results in the correct window. This cycle generally takes between 1.5 and 3 seconds.

C. Executive Dashboard and Reporting

Dashboard is generated on every load through the server. This design decision was made intentionally to ensure that the management gets updated data and not the data saved in the cache after visiting the web-page. There will be a graph showing the trends in production and defects during the previous 30 days, below these numbers.

Generation of the PDF document occurs through the server using a report generation library. PDF generation requests a database (PostgreSQL) to return all data according to the specified period and then generates the PDF document on the server side. For big periods, PDF generation may take some seconds; this issue is discussed in Section V.D. Development Process

We employed a sprint methodology for six sprints. The first sprint involved setting up the project, including creating the schema, provisioning the Neon database, and deploying a minimal version of the Next.js app on Vercel. Sprints two and three handled the Employee module and Production module, respectively. Sprint four dealt with the AI Command Center, which was the most challenging aspect of the project, particularly due to the number of attempts required to formulate prompts that would produce a valid JSON response. The fifth sprint added the executive dashboard visuals and PDF generation capability.

5. Results and Testing

A. Test Scenarios

We ran all tests against the live Neon Cloud database rather than a local instance, since that reflects actual deployment conditions. The scenarios below represent the main paths through the system.

Employee Lifecycle — The new employee was added in the system, assigned to a department, role update twice and finally soft-deleted. All four events were recorded by the AuditLog with timestamps for each operation. Response time for each API call was below 120 ms making it snappy enough.
Batch Record Insertion — 500 units worth of batch entry was submitted along with 12 defects logged in a batch. Efficiency rate of 97.6% was achieved by the system, and the revenue was computed using the unit price from the Product table and was inserted into the database in one go. The dashboard would reflect the data on the following page refresh.
Predictive Maintenance via AI — We made a query for 60 days of production records, formulated a prompt and sent the request to Groq. The AI model streamed out the maintenance report along with flags regarding two production lines that displayed high levels of defects in their operations. Time taken for all operations: 2.1 seconds.Anomaly Detection — Running the anomaly detection engine over the same 60-day window, the system flagged two batches where defect rates exceeded two standard deviations above the rolling mean. These showed up as highlighted alerts on the dashboard.
Natural language query - After inputting the question "Which department was the least efficient last week?", the system built a query with parameters and used PostgreSQL for its execution. The response was delivered in 1.8 seconds.
PDF export - The request to export a 90-day report was processed in 3.2 seconds and resulted in an accurately generated PDF file. The KPI values in the file coincided with those presented by the dashboard. Size of the file was approx. 400 KB.
The response times to the raw database queries on indexed columns varied from 20 to 80 milliseconds depending on how many tables the query involved – up to four and more, respectively.

Table 3: Performance Test Results

Feature / Test Scenario	Response Time	Observation
Employee CRUD Operations	< 120 ms	Full lifecycle management with complete audit trail logging
Production Batch Insert & KPI	< 200 ms	Real-time KPI recalculation on successful commit
AI Analytics (Groq Streaming)	1.5 – 3 sec	Streaming JSON response from LLaMA 3.3 70B inference
Executive Dashboard Load (SSR)	< 800 ms	Next.js 15 server-side rendering with response caching
Anomaly Detection (60-day)	< 2 sec	Statistical pattern analysis over historical production data
PDF Report Export	2 – 5 sec	Server-side generation; varies with dataset size
PostgreSQL Query (indexed)	20 – 80 ms	Prisma ORM type-safe query execution on Neon Cloud

B. Interface Screens

The landing page provides an introduction into what FactoryFlow is and how to begin using it. Once logged in, users access the Executive Dashboard, from which all the important KPIs can be seen immediately. The AI Command Center includes four tabs (one tab for each analytics engine), each of which contains an "Run Analysis" button as well as a window where the results will appear. There are no buttons on the Employee and Production pages other than those used to create/edit database entries.

C. Limitations

A few limitations became apparent during testing. First, the AI tools work much better when there's at least a month of production history in the database. With only a handful of records, the model produces generic recommendations rather than anything specific to the organization. Second, we built FactoryFlow for a single organization. It has no multi-tenant architecture, so running it for multiple clients would require separate deployments. Third, PDF export slows down noticeably when generating reports covering more than 10,000 production records. We had planned to move this to a background job queue but didn't get to it within the project timeline.

6. Conclusion

The core question behind this project was whether a well-designed DBMS genuinely makes a measurable difference in how a manufacturing organization operates. Based on our experience building and testing FactoryFlow, the answer is yes — though with some caveats.

Replacing scattered spreadsheets with a single normalized PostgreSQL database had an immediate practical effect. KPIs that previously took manual effort to compile were available instantly. The audit trail made it possible to trace any data change back to its source. Defect rates and efficiency scores that previously existed only in weekly summary emails became queryable in real time.

The AI layer added value beyond what we initially anticipated. The predictive maintenance engine in particular surfaced patterns that would have been easy to miss in a manual review. Whether those recommendations would hold up at scale, with more varied production data, is an open question — but the early results were encouraging.

Response times across the board were well within acceptable limits for a production tool. Core DBMS operations came in under 200 ms; the AI engines under 3 seconds. The dashboard loaded in under 800 ms on a standard connection. None of these numbers required special optimization beyond proper indexing and the use of server-side rendering for the initial page load.

Future work should address multi-tenant support, background job processing for large exports, and IoT integration to allow production data to be written directly from factory floor sensors rather than entered manually. Mobile access for floor operators would also be a practical next step.

Acknowledgments

We would like to thank Dr. B.K. Verma, Head of the Department of Computer Science and Engineering (AI & Data Science) at Panipat Institute of Engineering and Technology, Samalkha, for supervising this project and for pushing us to think more carefully about the database design during the early sprints.

References:
[1] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, 7th ed. Hoboken, NJ: Pearson, 2022.
[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts, 7th ed. New York: McGraw-Hill, 2020.
[3] Next.js Team, "Next.js Documentation — App Router, API Routes, Server Components," Vercel, 2024. [Online]. Available: https://nextjs.org/docs
[4] Prisma Team, "Prisma ORM Documentation — Schema, Migrations, Client," Prisma, 2024. [Online]. Available: https://www.prisma.io/docs
[5] Neon Team, "PostgreSQL 16 Documentation — Neon Serverless Postgres," Neon, 2024. [Online]. Available: https://neon.tech/docs
[6] Groq Team, "Groq AI API Documentation — LLaMA 3.3 70B, Streaming," Groq, 2024. [Online]. Available: https://console.groq.com/docs
[7] Research papers on AI-Driven Industrial DBMS, Predictive Analytics in Manufacturing, and Organizational Performance Impact of Database Systems, 2023.
.