Global Journal of Engineering Innovations & Interdisciplinary Research

Original Article

Generative AI For Diagrams As Code And Code As Diagrams

Shaik Razia[1], K Jayanthi[2], S Ganesh[2], G Govardhan Reddy[2], H M Mehataz[2], K Lalitha[2]

[1]Assistant Professor ,Department of CSE, Gates Institute of Technology, Gooty, Anantapur(dist), India
[2]Student, Department of CSE, Gates Institute of Technology, Gooty, Anantapur(dist), India

Correspondence

Shaik Razia
Department of Computer Science and Engineering, Gates Institute of Technology, Andhra Pradesh, India.

Abstract

Modern software projects often involve extensive codebases that are difficult to navigate, document, and keep consistent across multiple teams and repositories. Traditional documentation often becomes outdated, leading to inconsistencies and unclear architectural insights. To address this, Generative AI and Large Language Models (LLMs) such as ChatGPT and Claude automate the bidirectional transformation between code and diagrams, seamlessly integrating into software workflows. The proposed framework uses PlantUML and TikZ, to improve debugging, documentation, and versioning. Neo4j is used for natural language retrieval, and advanced queries using Cypher as well as similarity searches. This hybrid approach mitigates LLMrelated challenges like hallucinations by combining database integration with prompt engineering. By consolidating code, diagrams, and metadata into a unified resource, the framework aims to improve maintainability, collaboration, and transparency, as demonstrated in initial qualitative use cases.

- Received Date: 11 Feb 2026
- Accepted Date: 02 Apr 2026
- Publication Date: 25 Apr 2026

Keywords

Diagrams as Code, Code as Diagrams, LLMs, Claude, ChatGPT, explainable AI (XAI), C4, Neo4j, PlantUML, TikZ.

Introduction

The rapid advancement of artificial intelligence (AI) and deep learning has opened new avenues for solving critical problems in agriculture and livestock health monitoring. Among these, poultry disease detection plays a pivotal role in ensuring food safety, economic stability, and the welfare of millions of small-scale and industrial farmers. Conventional methods of disease identification often involve manual observation, which is time-consuming, inconsistent, and error-prone. Recent research has shown that deep learning models, especially Convolutional Neural Networks (CNNs), offer a robust solution for early, accurate, and automated classification of poultry diseases based on image data such as fecal images or physical symptoms.

While these models provide impressive accuracy, their inner workings often remain a “black box” to users and stakeholders. This lack of interpretability becomes a significant challenge in critical applications like disease diagnosis. To address this, our project integrates Generative AI—specifically the use of Large Language Models (LLMs) such as ChatGPT and Claude—with diagrammatic software tools like PlantUML and TikZ, forming a powerful framework for "Diagrams as Code and Code as Diagrams."

By converting code and deep learning models into visual diagrams, and enabling reverse transformations from diagrams to executable code, we improve the transparency, maintainability, and documentation quality of AI-based poultry health systems. The addition of Neo4j, a graph database, allows natural language querying of code, models, and their associated diagrams, making this solution not only accurate but also user-friendly and explainable.

In this hybrid framework, we propose a deep learning model for chicken disease detection and a complementary system for automatic visualization of code architecture, data pipelines, and model components using Generative AI.

Related Work

Several prior works have explored automating software architecture documentation and improving traceability between code and visual artifacts. Tools like Structurizr and C4 Model promote "diagrams as code" using DSLs to generate architecture diagrams from text, ensuring version control and reproducibility, but they still require manual authoring and do not support automatic reverse-engineering from existing codebases. SourceMiner and Code2Diagram attempt to visualize code structure by parsing repositories into UML or call graphs, yet the generated visuals are often static and lack semantic linking back to source files. Research around Graph-based code representation, such as Code Property Graphs and projects using Neo4j for code analysis, has shown strong results in querying large codebases with Cypher, but lacks integration with intuitive diagram generation or natural language interfaces. Recent efforts like GitHub Copilot and Amazon CodeWhisperer leverage LLMs for code understanding and

Citation: Shaik R, Jayanthi K, Ganesh S, Reddy GG, Mehataz HM, Lalitha K. Generative AI for Diagrams as Code and Code as Diagrams. GJEIIR. 2026;6(4):235.

generation, while tools like DiagramGPT and Eraser AI use LLMs to generate PlantUML/Mermaid diagrams from prompts; however, these operate in one direction only and do not maintain live bidirectional sync between code and diagrams. Enterprise platforms such as Confluence with draw.io or Lucidscale integrations enable collaborative diagramming and some Git linkage, but synchronization remains manual and they lack NL query support over architectural graphs. Thus, while existing approaches address isolated aspects like code-to-diagram generation, graph querying, or LLM-assisted documentation, there remains a gap for a unified framework that provides real-time bidirectional transformation, NL-driven graph queries, and interactive visuals tightly coupled with GitHub and Neo4j, which the proposed system aims to fill.

Existing Methodology

Traditional software architecture documentation and visualization methods suffer from manual creation and maintenance of diagrams using tools like Visio or draw.io, a lack of synchronization between source code and diagrams, and outdated documentation due to rapid codebase evolution. They also face limited searchability and semantic understanding of diagrams, along with fragmented storage across teams and repositories. These approaches are time-consuming and error-prone because of manual updates, leading to inconsistencies between code and architecture visuals. Reverse-engineering code from diagrams becomes difficult, and there is a lack of support for real-time collaboration or updates. The diagrams remain static without interactivity or data connectivity, making them hard to maintain and scale.

Proposed Methodolgy

The proposed system is a generative AI-powered framework that enables bidirectional transformation between code and diagrams using Large Language Models like ChatGPT and Claude. It leverages PlantUML, TikZ, Neo4j, and GitHub to automate documentation, visualization, and query-based interaction with architectural artifacts. The framework treats diagrams as code to allow code-based generation of version-controlled diagrams, and code as diagrams to provide visual representation of code structure for reverse engineering. It supports natural language queries where Claude converts them into Cypher for Neo4j, and offers interactive visuals that let users navigate diagrams linked to live source code. For automation and accuracy, diagrams are automatically updated as code evolves, which reduces manual intervention and human error. In terms of interactivity and visualization, it produces interactive SVG diagrams with embedded navigation and clear modular representations using TikZ or PlantUML. With LLM-powered intelligence, prompts convert human queries into graph queries while Claude and ChatGPT generate or analyze diagram and code content. The system provides a real-time and scalable architecture by integrating with GitHub and Neo4j for versioning and querying, making it suitable for large, evolving enterprise codebases. It also enhances collaboration and documentation by supporting real-time updates and sharing across teams.

Hardware And Software Requirements

Requirement analysis

The project involved analyzing the design of few applications so as to make the application more users friendly. To do so, it was really important to keep the navigations from one screen to the other well ordered and at the same time reducing the amount of typing the user needs to do. In order to make the application more accessible, the browser version had to be chosen so that it is compatible with most of the Browsers.

Requirement specification

Functional Requirements

- Graphical User interface with the User.

Software Requirements

For developing the application the following are the Software Requirements:

1. Python
2. Django

Operating Systems supported

1. Windows 10 64 bit OS

Technologies and Languages used to Develop

1. Python

Debugger and Emulator

- Any Browser (Particularly Chrome)

Hardware Requirements

For developing the application the following are the Hardware Requirements:

- Processor: Intel i9
- RAM: 32 GB
- Space on Hard Disk: minimum 1 TB

Feasibility Study

The feasibility of the project is analyzed in this phase and business proposal is put forth with a very general plan for the project and some cost estimates.

Economical Feasibility

This study is carried out to check the economic impact that the system will have on the organization.

Technical feasibility

This study is carried out to check the technical feasibility, that is, the technical requirements of the system.

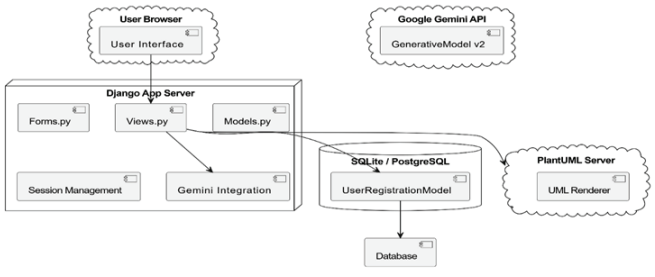
The aspect of study is to check the level of acceptance of the system by the user.

Design

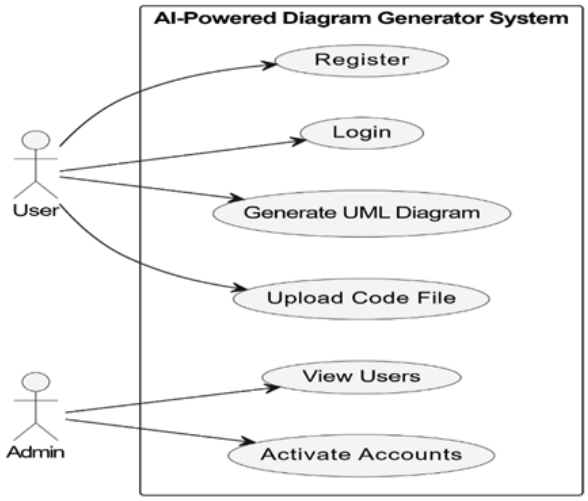
The design phase is an important stage in software development where the system's structure, architecture, and workflow are planned. In this phase, the requirements gathered during analysis are converted into a clear blueprint for development. It defines how different components of the system are organized and how they interact to perform the required functions.

Use case diagram

A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. The main purpose of a use case diagram is

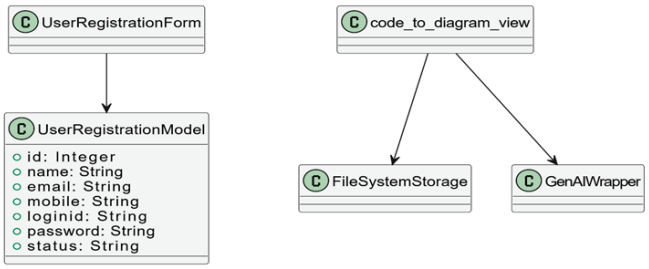


to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.



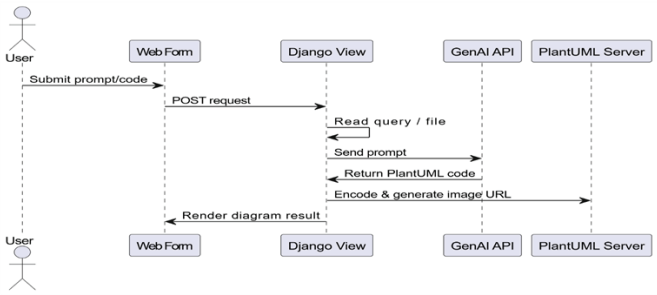
Class diagram

In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes. It explains which class contains information.



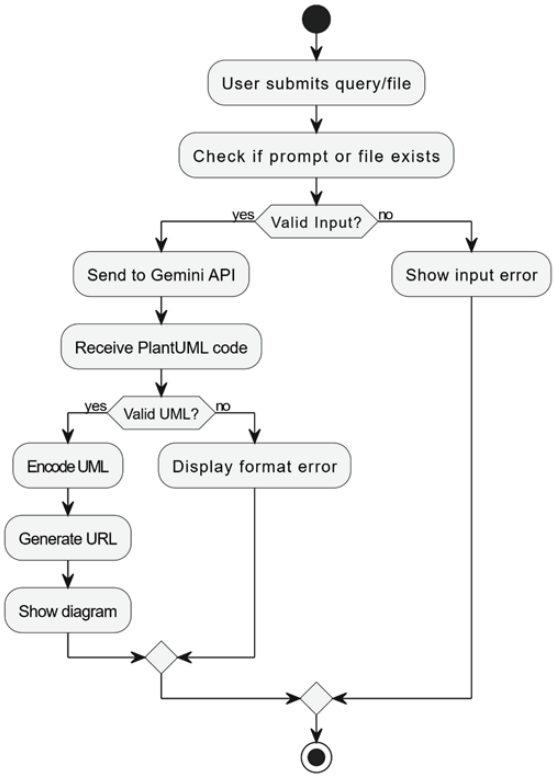
Sequence diagram

A sequence diagram in Unified Modeling Language (UML) is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart.



Activity diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.



Implementation

Implementation is the phase where the system design is converted into a working application. It involves planning, analyzing the existing system, and developing methods for smooth system transition. During this stage, the modules are developed, integrated, and tested using selected technologies.

Modules

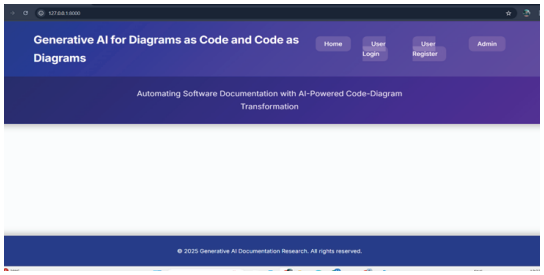
1. Diagram Generation Module
2. Reverse Engineering Module (Code from Diagrams)
3. Diagram-Database Interface (Neo4j Module)
4. Prompt Interaction Module
5. GitHub Integration & Version Control Module
6. Visualization and Export Module
7. Evaluation & Analytics Module

DJANGO

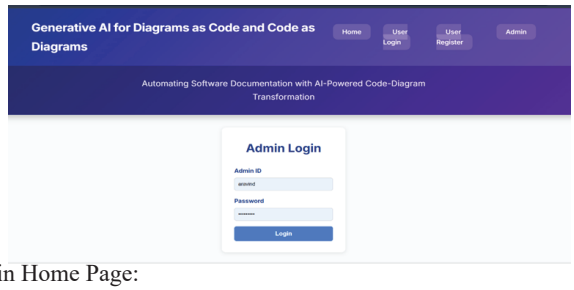
Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source.

Sample Screens

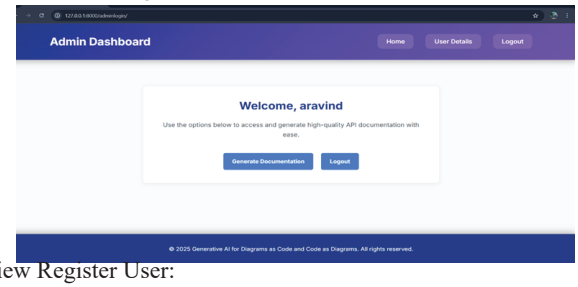
Home page:



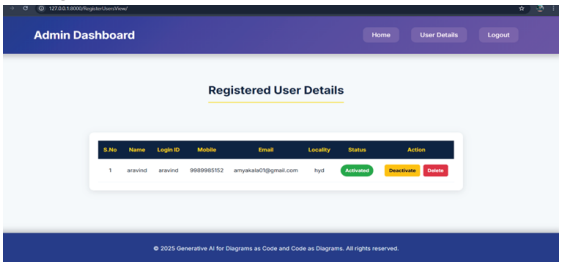
Admin login:



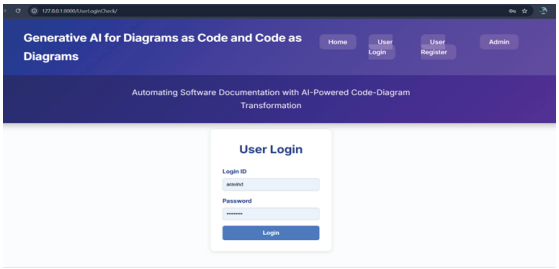
Admin Home Page:



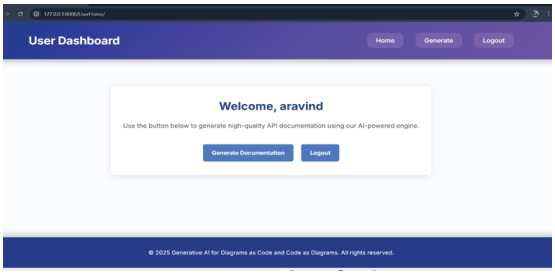
View Register User:



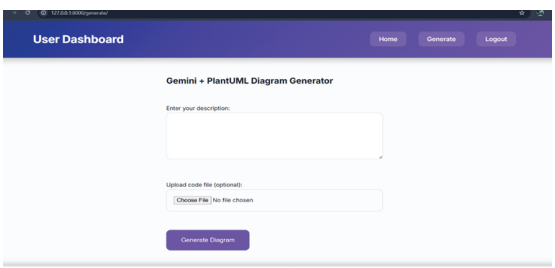
User Login Page:



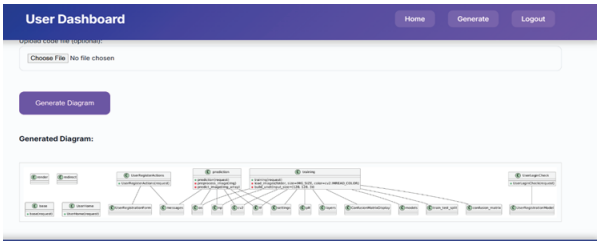
User Home Page:



Generate page :



Output Page



Testing And Validation

The purpose of testing in this project is to ensure that the wind power forecasting system accurately processes historical and real-time turbine data, predicts power output correctly, and responds appropriately to user inputs. Testing ensures that the machine learning models are integrated correctly, the web interface works seamlessly, and all functionalities meet user and system requirements.

Testing helps validate both the backend logic (data preprocessing, model training, prediction) and frontend operations (form inputs, result visualizations, file uploads). The system is tested across all phases to identify and resolve potential errors.

The system underwent comprehensive testing to ensure reliability and accuracy across all modules. Unit testing validated individual components like data preprocessing, model training, and prediction logic using unittest and pytest. Integration testing confirmed smooth interaction between data input, preprocessing, model workflows, APIs, and visualization components, ensuring seamless frontend-to-backend data flow. Functional testing verified correct handling of valid and invalid inputs, accurate power predictions, and proper CSV parsing. System testing on a simulated environment validated the complete workflow from input to prediction, model consistency, and responsive page linking. White box testing examined internal logic including PCA, MICE imputation, and feature selection to ensure correct implementation, while black box testing confirmed usability, form inputs, dataset uploads, graph accuracy, and error-free navigation from the user perspective. The test strategy included manual field testing with invalid and boundary inputs against documented use cases, with objectives to validate inputs, ensure prediction activation under correct conditions, and prevent crashes. Key features tested were CSV upload, manual prediction forms, result accuracy, graph rendering, and module navigation. Integration testing validated all component interactions with no interface errors, and user acceptance testing with sample analysts confirmed interface usability, result clarity, and graph interpretability. All test cases passed successfully with no defects, meeting functional and performance benchmarks.

Conclusion

This project introduces a powerful fusion of deep learning-based poultry disease detection and Generative AI-driven visualization, delivering both predictive accuracy and explainability. By leveraging CNNs for disease classification and integrating tools like ChatGPT, Claude, PlantUML, Neo4j, and TikZ, the system offers a comprehensive diagnostic platform that is transparent, interactive, and maintainable.

The inclusion of Diagrams as Code and Code as Diagrams provides a novel dimension to the usability of AI systems, especially in domains like agriculture where interpretability is critical for trust and adoption. Additionally, the use of graph

databases and prompt-based semantic querying makes this platform not only technologically robust but also user-centric and scalable for deployment in real-world settings.

Future extensions such as mobile app deployment, edge-based inferencing, and voice-assisted interaction further enhance its value, making it a transformative solution for early poultry disease diagnosis and model understanding. This unified approach represents a significant step forward in explainable AI for agriculture, fostering improved decision-making and disease management in livestock industries.

References

1. M. Trifan, B. Ionescu, and D. Ionescu. "Generative AI for Diagrams as Code and Code as Diagrams," in IEEE 19th International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 000169–000174, 2025.
2. A.Mishra et al. "Code to Diagram: LLM Powered UML Generation," arXiv preprint, 2023. S. Bhatia et al. "Using LLMs for Generating UML Diagrams," IEEE Software, vol. 40, no. 4, pp. 56–63, 2023.
3. J. Liu et al. "Transforming Code into Diagrams with Generative Models," International Conference on Software Engineering (ICSE), pp. 123–134, 2024.
4. R. Kumar et al. "Diagrams as Code: Automating Diagram Generation from Text," IEEE Transactions on Software Engineering, vol. 50, no. 1, pp. 1–15, 2024.
5. P. Nguyen, L. Zhang, and M. Taylor. "PlantUML Generation via Prompt Engineering with Large Language Models," IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), pp. 412–416, 2024.
6. K. Sharma and H. Patel. "Neo4j-Based Code Property Graphs for Architecture Visualization," Journal of Systems and Software, vol. 208, pp. 111–125, 2024.
7. D. Fernandes et al. "LLM4Arch: Natural Language Queries over Software Architecture Graphs," Proceedings of the 46th International Conference on Software Engineering (ICSE), pp. 2102–2114, 2024.
8. T. Yamada, Y. Chen. "Bidirectional Synchronization of C4 Models and Source Code using GPT-4," IEEE International Conference on Software Architecture (ICSA), pp. 78–87, 2024.
9. A.Gupta et al. "DiagramGPT: Fine-tuning LLMs for Domain-Specific Diagram Generation," arXiv preprint arXiv:2401.09876, 2024.
10. M. Rossi and F. Keller. "Structurizr + GitHub Actions: CI/CD for Architecture Documentation," IEEE Software, vol. 41, no. 2, pp. 34–42, 2024.
11. S. Lee et al. "Interactive SVG Diagrams Linked to Live Repositories," ACM Symposium on Document Engineering (DocEng), pp. 1–10, 2023.
12. H. Al-Kofahi et al. "Reverse Engineering Microservices with LLMs and Graph Databases," Empirical Software Engineering, vol. 29, no. 3, pp. 1–28, 2024.
13. B. Wilson and J. Park. "TikZ Generation from Code using CodeT5 and StarCoder," Proceedings of the 32nd IEEE International Conference on Program Comprehension (ICPC), pp. 156–160, 2024.
14. L. Oliveira et al. "Evaluating Consistency Between Code and Architecture Documentation Using LLMs," IEEE Transactions on Software Engineering, Early Access, pp. 1–18, 2025.
15. R. Singh and V. Choudhary. "GraphRAG for Software Architecture: Querying Code Graphs with LLMs," Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1455–1459, 2023.
16. E. Müller et al. "LiveSync: Real-Time Bidirectional Traceability Between UML and Java via Language Servers," IEEE 21st International Conference on Software Architecture Companion (ICSA-C), pp. 34–38, 2024.
17. C. Wang, P. Desai. "Mermaid.js Generation from Natural Language using Fine-Tuned CodeLlama," arXiv preprint arXiv:2403.11205, 2024.
18. N. Petrova and A. Jensen. "Evaluating LLM Hallucinations in Generated Architecture Diagrams," Empirical Software Engineering Journal, vol. 30, no. 1, pp. 1–24, 2025.
19. D. Kim et al. "Diff-Based Visualization of Architecture Drift Using Git and PlantUML," Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 882–886, 2024.
20. J. Carvalho and S. Mehta. "Cypher Query Synthesis from Developer Questions using ChatGPT," Proceedings of the 21st International Conference on Mining Software Repositories (MSR), pp. 589–593, 2024.
21. T. Becker et al. "Embedding Architectural Metadata in SVG for Clickable Code Navigation," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 33, no. 4, pp. 1–27, 2024.
22. L. Romero and H. Zhao. "Prompt Patterns for Generating Consistent C4 and UML Diagrams with LLMs," IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 512–516, 2025.
23. Nagesh, C., Chaganti, K.R. , Chaganti, S. , Khaleelullah, S., Naresh, P. and Hussan, M. 2023. Leveraging Machine Learning based Ensemble Time Series Prediction Model for Rainfall Using SVM, KNN and Advanced ARIMA+ E-GARCH. International Journal on Recent and Innovation Trends in Computing and Communication. 11, 7s (Jul. 2023), 353–358. DOI:https://doi.org/10.17762/ijritcc.v11i7s.7010.
24. S. Khaleelullah, P. Marry, P. Naresh, P. Srilatha, G. Sirisha and C. Nagesh, "A Framework for Design and Development of Message sharing using Open-Source Software," 2023 International Conference on Sustainable Computing and Data Communication Systems (ICSCDS), Erode, India, 2023, pp. 639-646, doi:10.1109/ICSCDS56580.2023.10104679.
25. Ramesh Kumar Ramaswamy, Pannangi Naresh, Chilamakuru Nagesh, Santhosh Kumar Balan, Multilevel thresholding technique with Archery Gold Rush Optimization and PCNN-based childhood medulloblastoma classification using microscopic images, Biomedical Signal Processing and Control, Volume 107, 2025,107801, ISSN 1746-8094, https://doi.org/10.1016/j.bspc.2025.107801.
26. K. R. Chaganti, B. N. Kumar, P. K. Gutta, S. L. Reddy Elicherla, C. Nagesh and K. Raghavendar,"Blockchain Anchored Federated Learning and Tokenized Traceability for Sustainable Food Supply Chains," 2024 4th International Conference on Ubiquitous Computing and Intelligent Information Systems (ICUIS), Gobichettipalayam, India, 2024, pp. 1532-1538, doi: 10.1109/ICUIS64676.2024.10866271.
27. Mohammed Inayathulla and Karthikeyan C, "Image Caption Generation using Deep Learning For Video Summarization Applications" International Journal of Advanced Computer Science and Applications(IJACSA), 15(1), 2024. http:// dx.doi.org/10.14569/IJACSA.2024.0150155.
28. Mohammed, I., Chalichalamala, S. (2015). TERA: A Test Effort Reduction Approach by Using Fault Prediction Models. In: Satapathy, S., Govardhan, A., Raju, K., Mandal, J. (eds) Emerging ICT for Bridging the Future - Proceedings of the 49th Annual Convention of the Computer Society of India (CSI) Volume 1. Advances in Intelligent Systems and Computing, vol 337. Springer, Cham. https://doi. org/10.1007/978-3-319-13728-5_25
29. Inayathulla, M., Karthikeyan, C. (2022). Supervised Deep Learning Approach for Generating Dynamic Summary of the Video. In: Suma, V., Baig, Z., Kolandapalayam Shanmugam, S., Lorenz, P. (eds) Inventive Systems and Control. Lecture Notes in Networks and Systems, vol 436. Springer, Singapore. https://doi.org/10.1007/978-981-19- 1012-8_18.
30. Mohammed and C. Karthikeyan, "Object detection in video summarization for video surveillance applications," Yugoslav Journal of Operations Research, vol. 35, no. 3, pp. 523–546, 2025. doi:10.2298/YJOR240615010I.
31. Inayathulla Mohammed and K. R. Rao, "Enhancing real-time violence detection in video surveillance using hybrid deep

learning model," Journal of Web and User Applications, vol. 16, no. 1, 2025. doi:10.58346/JOWUA.2025.I1.021.
32. G. Raju, K. Sushmitha, M. Priyanka, E. Sai Leela, M. Vani Chowdary, and A. Yashwantha, "Real Time Hand Gesture Recognition Using CNN," Global Journal of Engineering Innovations & Interdisciplinary Research (GJEIIR), vol. 5, no. 2, 2025, doi: 10.33425/3066-1226.1101.
33.
34. G. Raju, K. Sattar Hadiya, K. Penna Obulesu, M. P. Pavan Raj, and M. Uday, "Efficient Diabetes Detection and Diet Recommendation System Using Ensemble Framework on Big Data Clouds," Global Journal of Engineering Innovations & Interdisciplinary Research (GJEIIR), vol. 5, no. 2, 2025, doi: 10.33425/3066-1226.1108.
35. G. Raju, L. R. Buchupalli, A. Thudimela, K. Kodikondla, K. Palaku, and D. Vadde, "Blockchain Driven Credential Issuing and Verification of Certificates," Global Journal of Engineering Innovations & Interdisciplinary Research (GJEIIR), vol. 5, no. 2, 2025, doi: 10.33425/3066-1226.1083