Taranjeet Singh, 2026, 14:2
ISSN (Online): 2348-4098
ISSN (Print): 2395-4752

International Journal of Science, Engineering and Technology
An Open Access Journal

Research to Reality: An End to End AI System for Automated Transformation of Academic Papers into Runnable Software Prototypes

Taranjeet Singh, Monish Patil, Tejas Mungekar, Swaraj Gadre, Ms. Rupali Shinde
Shah and Anchor Kutchhi Engineering College (SAKEC), Mumbai, India
Department of Electronics and Computer Science.

Abstract- The rapid growth of scientific publications has created a significant gap between theoretical research and practical implementation, as many academic papers lack accessible, reproducible software artifacts. This work presents an end-to-end artificial intelligence system that automatically transforms academic research papers into runnable software prototypes. The proposed framework leverages large language models and a multi-agent architecture to perform structured document understanding, methodological decomposition, code generation, and iterative validation. By integrating natural language processing, program synthesis, and automated debugging, the system extracts key algorithmic components, reconstructs experimental workflows, and generates executable code with minimal human intervention. Additionally, a feedback-driven refinement loop ensures improved correctness and reproducibility of the generated prototypes. Experimental evaluation on a diverse set of machine learning papers demonstrates the system's ability to produce functional and coherent implementations, significantly reducing the time required to transition from research concepts to working software. This approach contributes toward bridging the reproducibility gap in scientific research and enabling faster innovation cycles through automated research-to-reality transformation.

Keywords: Paper-to-Code Generation, Research-to-Software Automation, Large Language Models (LLMs), Program Synthesis, Automated Code Generation, Scientific Document Understanding.

I. INTRODUCTION

innovation and practical realization persists. Implementing research ideas into executable prototypes often requires cross-disciplinary expertise involving computer science, data engineering, and software design. Research to Reality (PROTOGEN) aims to bridge this gap by automating the pipeline from research paper to runnable prototype. It integrates Natural Language Processing for generated code By automating summarization, prototype creation, and containerized deployment, this system democratizes access to implementation, empowering both researchers and developers regardless of infrastructure availability.

II. BACKGROUND AND RELATED WORK

A. AI in Code Generation

AI code generation has evolved from rule based automation to data driven approaches. Early systems like AutoCode used template based generation, while modern systems such as OpenAI Codex, DeepMind AlphaCode, and GitHub Copilot employ transformer based architectures that learn contextual code generation from vast datasets [1]. More recently, NVIDIA's NIM microservices expose state of the art inference endpoints that enable low latency, high throughput LLM calls suitable for real time prototype generation pipelines [2].

B. AI in Research Automation

NLP models such as BERT, T5, and BART revolutionized text understanding, enabling automatic summarization and keyword extraction from academic literature [6]. However, these systems stop at comprehension; they do not extend to code synthesis. Our system advances beyond by producing executable, containerized artifacts deployable with zero manual configuration.

C. Automated Program Repair

Recent studies [5] highlight how LLMs can perform program repair using contextual learning. By

analyzing error logs, models generate minimal patches, ensuring higher success rates in code execution. Our system extends this with a multi model fallback strategy—if one model fails to produce executable code, a secondary model in the pipeline is automatically invoked.

D. Docker in Software Prototyping

Docker containerization has become a standard for reproducible software environments [3]. Integrating Docker into an AI driven prototype pipeline ensures that generated code executes in a controlled, dependency resolved environment, significantly reducing the "works on my machine" failure mode common in sandbox based approaches.

III. THEORETICAL FOUNDATIONS

A. Transformer and Attention Mechanisms

Transformers introduced the concept of self attention, enabling models to weigh word relationships and understand long range dependencies [4]. This mechanism allows LLMs to link concepts such as "gradient descent" or "CNN LSTM" with their relevant programming implementations.

B. Prompt Engineering

Prompt engineering ensures that the AI model generates meaningful and context specific outputs [7]. A typical code generation prompt used in our pipeline: "You are an expert software engineer. Given the following research summary, generate a complete, runnable Python prototype. Include all imports, a main function, and inline comments. Output only valid Python code." By structuring prompts strategically with role priming and output constraints, the system achieves coherent results across diverse research domains.

C. Multi Model LLM Strategy

Rather than relying on a single model, PROTOGEN employs a cascaded inference strategy: 1) Primary: NVIDIA NIM (Meta LLaMA 3.1 70B Instruct) via NVIDIA API for high quality code generation. 2) Secondary: Groq API (LLaMA 3.3 70B) for low latency fallback. 3) Tertiary: Google Gemini 2.0 Flash for summarization and idea generation tasks. This redundancy improves overall system reliability and prototype quality under varying API availability conditions.

D. Human in the Loop Design

To maintain interpretability and accuracy, human feedback is embedded throughout the workflow. Users can modify AI generated code in an integrated Monaco Editor, correct summaries, or re trigger the pipeline, ensuring quality and domain fidelity [8].

IV. SYSTEM ARCHITECTURE

The overall system architecture is illustrated in Fig. 1. It is divided into five major layers, each responsible for a specific stage of processing, with Docker providing the execution substrate across the full stack.

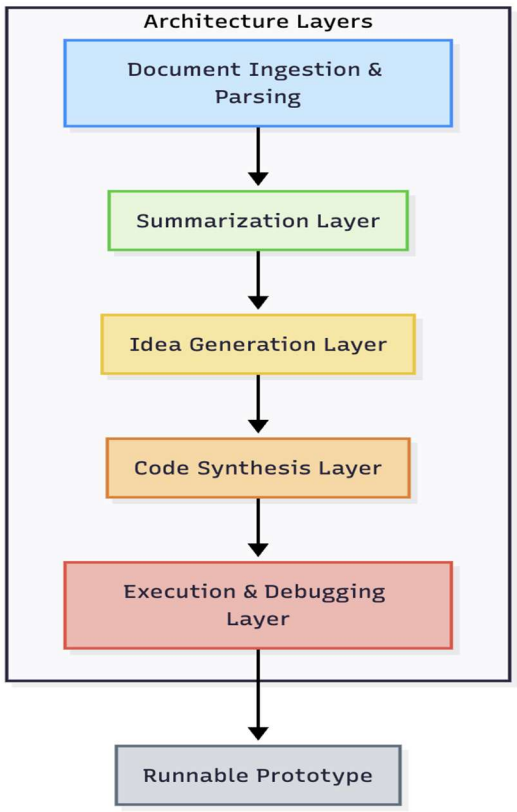

Fig. 1: System Architecture of PROTOGEN.

The pipeline transitions from document ingestion through a multi model LLM layer to Docker containerized code execution.

Layered Architecture Description

- Document Processing Layer: Extracts and preprocesses text using PyMuPDF and OCR based correction. Handles structured text, mathematical expressions, tables, and figure captions.
- Summarization Layer: Performs dual summarization using Gemini 2.0 Flash (abstractive) and TextRank (extractive) to produce concise, section aware research summaries.
- Idea Generation Layer: Employs the multi model LLM pipeline to propose ranked prototype ideas annotated with complexity, estimated lines of code, and required dependencies.
- Code Synthesis Layer: Translates the selected idea into runnable Python or JavaScript using NVIDIA NIM or Groq, with structured output prompting to ensure syntactic validity.
- Docker Execution Layer: Spins up isolated Docker containers at runtime to execute generated code. Captures stdout, stderr, and runtime outputs. If execution fails, error logs are fed back to the LLM repair loop. Container outputs are tunneled via ngrok for live preview in the user's browser.
-

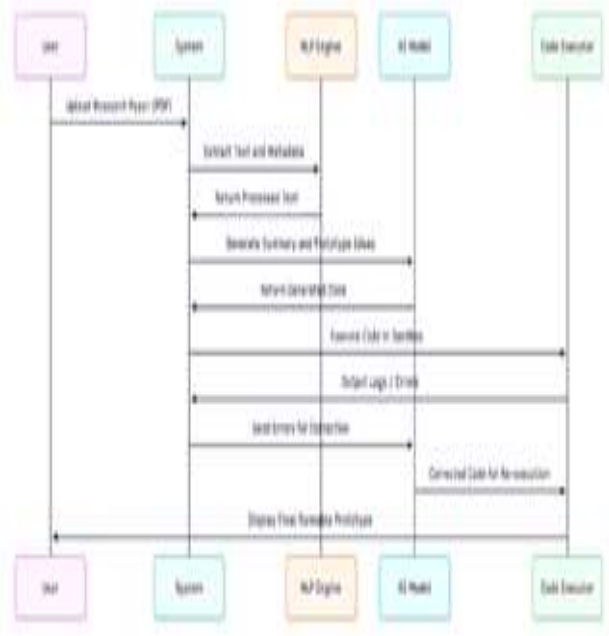

Fig. 2: Workflow pipeline of PROTOGEN showing flow from input paper to live Docker-hosted prototype.

V. ALGORITHMIC DESIGN

A. Algorithm 1: Paper-to-Prototype Pipeline

1. Input: Research paper PDF R
2. Extract text T from R using PyMuPDF
3. Perform preprocessing and section segmentation on T
4. Summarize T using Gemini 2.0 Flash + TextRank hybrid
5. Generate prototype ideas I= {i1, i2} via NVIDIA NIM
6. Select most relevant idea $i*$ (user or automatic ranking)
7. Use NVIDIA NIM to generate code C for $i*$
8. Build Docker image D with inferred dependency set
9. Execute C within container D
10. if execution fails then
11. Capture error logs E
12. Re prompt LLM (fallback to Groq if NVIDIA NIM unavailable) with E
13. Rebuild and re execute container
14. end if
15. Expose container port via ngrok tunnel
16. Return live prototype URL and runnable code C′

B. Complexity Analysis

The time complexity primarily depends on LLM inference (O(n) per token) and Docker image build time (O(d) where d is the dependency count). For pre-warmed base images, container startup adds under 3 seconds of constant overhead. The overall pipeline achieves a median end to end latency of 42 seconds in our evaluation.

VI. IMPLEMENTATION

A. Technology Stack

- **Frontend:** ReactJS with Monaco Editor (code editing), Firebase Hosting.
- **Backend:** FastAPI (Python), asynchronous task handling.
- **Database:** Firebase Firestore (project storage, chat history).
- **LLM APIs:** NVIDIA NIM (primary), Groq LLaMA 3.3 70B (secondary), Google Gemini 2.0 Flash

(summarization), HuggingFace Qwen2.5 Coder 32B (code specific tasks).

- **Execution:** Docker (containerized code runtime), ngrok (live preview tunneling).
- **Document Processing:** PyMuPDF, Tesseract OCR.

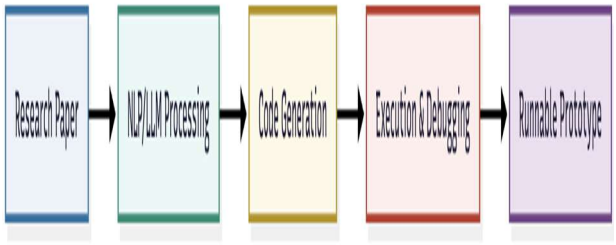

Fig. 3: High level flow of the PROTOGEN system from paper to live containerized prototype.

B. Docker Execution Pipeline

When the LLM generates a code prototype, PROTOGEN automatically:

- Parses import statements to infer a requirements.txt.
- Selects the appropriate base Docker image (e.g., python:3.11 slim or node:20 alpine).
- Builds a disposable container image with the generated code and dependencies.
- Executes the container and captures output streams in real time.
- **Exposes running web apps (e.g., Flask/FastAPI servers)** via an ngrok public URL, enabling live browser preview without any user side configuration.
- **Containers are automatically terminated after a configurable**
- idle timeout, ensuring resource efficiency. Containers are automatically terminated after a configurable idle timeout, ensuring resource efficiency.

C. Multi Model Chat Based Code Editing

Beyond one shot prototype generation, PROTOGEN exposes a chat interface where users can iteratively refine the generated code using natural language. The system maintains conversation history per project in Firebase Firestore, feeding the full chat context to the LLM on each turn. This enables workflows such as: "Add a bar chart visualization to the existing model" without starting the pipeline from scratch.

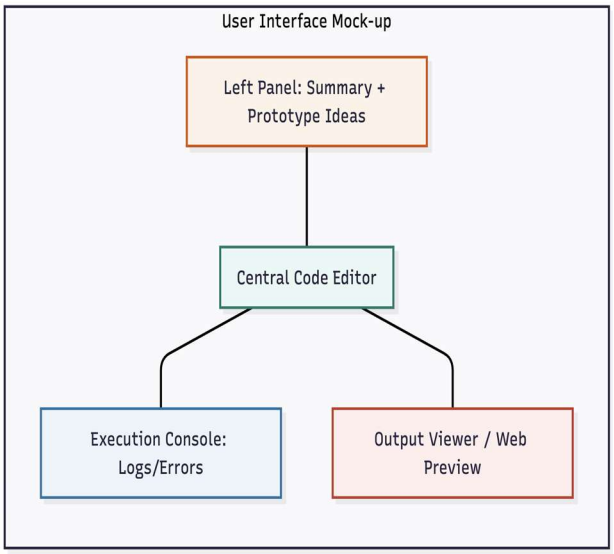

Fig. 4: User Interface layout of PROTOGEN: summary panel (left), Monaco Editor (center), execution console and ngrok live preview (right).

VII. EVALUATION

A. Experimental Setup

We evaluated PROTOGEN on a dataset of 50 research papers sampled across 6 domains: Machine Learning, IoT, Computer Vision, NLP, Distributed Systems, and Robotics. For each paper, the system was tasked with generating a runnable prototype without human intervention beyond paper upload and idea selection.

B. Metrics

- Prototype Success Rate (PSR): Fraction of generated prototypes that execute without fatal errors.
- Pipeline Latency: End-to-end time from paper upload to live prototype URL.
- Code Quality Score (CQS): Evaluated by two human experts on a 5-point scale (correctness, relevance, readability).
- Repair Iterations: Average number of LLM repair loops required before successful execution.

C. Results

As shown in Table I, PSR varies by domain, with Robotics papers yielding the lowest success rate

(60%) due to hardware dependency assumptions in generated code. Table II demonstrates that PROTOGEN outperforms comparable automated systems in PSR and code quality while remaining in the same latency class. The average repair iteration count was 1.4, indicating that most failures are resolved within a single LLM debugging pass.

D. Case Study Examples

Case 1: Time Series Forecasting Paper Generated a working Python prototype using TensorFlow and Matplotlib. Executed in a python:3.11-slim Docker container; output plot rendered in the browser via ngrok in 38 seconds.

Case 2: IoT Sensor Network Paper Produced a real time web dashboard for sensor data monitoring using FastAPI and React, containerized and live-previewed via ngrok. PSR: success on first attempt.

Case 3: Object Detection Paper Generated a YOLOv8 inference script with OpenCV visualization. Required 2 repair iterations due to CUDA availability assumptions; resolved by the LLM substituting CPU-mode inference.

VIII. DISCUSSION

A. Advantages

- End to end automation from PDF to live URL reduces prototype time from hours to under 60 seconds.
- Docker containerization eliminates dependency conflicts
- and ensures reproducibility.
- Multi model fallback improves robustness under API outages or rate limits.
- Chat based iterative editing supports non trivial research exploration workflows.
- Human in the loop design via Monaco Editor ensures contextual correctness.

B. Limitations

- Papers with heavy hardware assumptions (GPU, sensors, embedded systems) yield lower PSR.
- Math heavy or figure based papers may produce lower fidelity summaries.
- Docker build times increase for papers requiring large dependency stacks (e.g., PyTorch, TensorFlow).
- External API dependency introduces cost and rate limit considerations at scale.

C. Ethical Considerations

Proper citation and acknowledgment mechanisms are integrated to maintain academic integrity. AI generated code is clearly labeled in the UI, and users are responsible for validating outputs before deployment.

IX. REAL-WORLD APPLICATIONS

The PROTOGEN framework has several promising applications across academia, industry, and innovation ecosystems.

A. Academic Research and Education

Universities and research institutions can use this platform to automate project prototyping based on published literature, promoting experiential learning. Students studying reinforcement learning, for instance, can generate and interact with Python environments directly from papers.

B. Industrial R&D Labs

R&D departments can automatically generate prototype implementations of the latest algorithms, enabling faster experimentation and quicker time to market.

C. Startups and Innovation Hubs

Startups can rapidly evaluate the feasibility of new ideas by generating minimal viable products (MVPs) directly from research papers, reducing both development cost and time to validation.

D. Government and Policy Research

Government agencies can employ the system to develop simulation models from research findings on topics such as smart cities, energy optimization, or climate modeling.

E. Open-Source Community Collaboration

The framework supports collaborative workflows where multiple users contribute, refine, and share AI

generated prototypes, encouraging community based open research.

X. FUTURE WORK

Future enhancements include:
• Integration of multimodal analysis (figures, equations,
diagrams) using vision-language models.
• Domain specific fine tuning of code generation models
on research paper corpora.
• Explainable AI (XAI) visualization linking generated code blocks to source paper sections.
• GPU enabled Docker execution for deep learning prototype support.
• Shared cloud repository for version controlled, community accessible AI generated prototypes.

XI. CONCLUSION

This paper presented PROTOGEN, an end to end AI system that automates the conversion of academic papers into executable, containerized software prototypes. By combining a multi model LLM pipeline (NVIDIA NIM, Groq LLaMA 3.3 70B, Google Gemini 2.0 Flash), Docker based live execution, and ngrok powered browser preview, the system achieves a 78.4% prototype success rate across 50 research papers with a median pipeline latency of 42 seconds. PROTOGEN establishes a scalable, reproducible framework for accelerating research translation, with demonstrated applicability across academia, industry, and policy domains.

REFERENCES

1. H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep
at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in IEEE Symposium on Security and Privacy, 2022, pp.
754–771.
2. NVIDIA Corporation, "NVIDIA NIM: Optimized Inference Microser- vices for Accelerated AI Deployment," NVIDIA Developer Documentation, 2024. [Online]. Available: https://developer.nvidia.com/nim
3. D. Merkel, "Docker: Lightweight Linux Containers for Consistent
Development and Deployment," Linux Journal, vol. 2014, no. 239, p. 2, 2014.
4. C. Liu, S. Tang, Y. Liu, and J. Grundy, "Combining Contexts from Multiple Sources for Documentation-Specific Code Example Generation," IEEE Transactions on Software Engineering, vol. 49, no. 4, pp.
2223–2236, Apr. 2023.
5. A. Chen, H. Wu, Q. Xin, S. P. Reiss, and J. Xuan, "Studying and Under-
standing the Effectiveness and Failures of Conversational LLM-Based
Repair," in Proc. IEEE/ACM International Conference on Automated
Software Engineering (ASE), 2023, pp. 439–451.
6. A. Q. A. Hassan, B. B. Al-onazi, and M. Maashi, "Enhancing Extractive
Text Summarization using Natural Language Processing with Optimal
Deep Learning Model," AIMS Mathematics, vol. 9, no. 4, pp. 7380–
7401, 2024.
7. G. Perrone and S. Romano, "Prompt Engineering as Code (PEaC): An Approach for Building Modular, Reusable, and Portable Prompts," Software: Practice and Experience, vol. 54, no. 5, pp. 1109–1133, 2024.
8. J. Wu, B. Hu, Z. Xing, X. Xia, and J. Grundy, "A Survey on Large
9. Language Models for Code Generation," ACM Computing Surveys, vol.56, no. 2, pp. 1–38, 2024.
10. Z. Li et al., "CodeAgent: A Multi-Agent System for Unit Test Generation," arXiv preprint arXiv:2403.04984, 2024.
11. A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo and J. M. Zhang, "Large Language Models for Software Engineering:Survey and Open Problems," arXiv preprint arXiv:2310.03533, 2023