



From Paper to Prototype: An End-to-End AI System for Automated Research Implementation

Taranjeet Singh¹, Monish Patil², Tejas Mungekar³, Swaraj Gadre⁴, Rupali Shinde⁵

Students, Department of Electronics & Computer Science, Shah & Anchor Kutchhi Engineering College¹⁻⁴

Department of Electronics & Computer Science, Shah & Anchor Kutchhi Engineering College⁵

Abstract: Bridging the gap between academic research and real-world implementation remains a complex challenge in the technological ecosystem. While research papers propose algorithms and frameworks, converting them into runnable prototypes requires significant time, effort, and programming knowledge. This paper presents Research to Reality (PROTOGEN), an intelligent, end-to-end AI-driven system that automates the translation of academic research into executable software. By leveraging Natural Language Processing (NLP), a multi-model Large Language Model (LLM) pipeline powered by NVIDIA NIM, Groq (LLaMA-3.3-70B), and Google Gemini 2.0 Flash, the system performs text extraction, summarization, ideation, code generation, and live execution via Docker containerization. The system achieves a prototype generation success rate of 78.4% across 50 research papers spanning 6 domains, with an average end-to-end pipeline latency of 42 seconds. The work combines theoretical understanding with practical implementation, illustrating how AI systems can accelerate innovation, enhance collaboration, and make research outputs more actionable.

Keywords: AI code generation, NLP, LLM, software prototyping, research automation, human-in-the-loop, program synthesis, Docker containerization, NVIDIA NIM, multi-model pipeline

I. INTRODUCTION

Academic research has long served as the foundation for technological progress. However, the gap between theoretical 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 text understanding, a multi-model LLM pipeline for code synthesis, and Docker-based live execution for isolated, reproducible prototype deployment. The primary goal is to reduce manual intervention and enhance the speed and efficiency of research translation.

II. Motivation

Traditional research implementation is often hindered by the following challenges:

1. Lack of coding expertise among researchers.
2. Time-intensive manual translation of theoretical ideas.
3. Misinterpretation of research intent by developers.
4. Absence of reproducible execution environments 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. PROTOGEN 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. PROTOGEN 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 the pipeline instructs the model to act as an expert software engineer and, given a research summary, generate a complete, runnable Python prototype with all imports, a main function, and inline comments, outputting 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 divided into five major layers, each responsible for a specific stage of processing, with Docker providing the execution substrate across the full stack.

A. Layered Architecture Description

1. Document Processing Layer: Extracts and preprocesses text using PyMuPDF and OCR-based correction. Handles structured text, mathematical expressions, tables, and figure captions.
2. Summarization Layer: Performs dual summarization using Gemini 2.0 Flash (abstractive) and TextRank (extractive) to produce concise, section-aware research summaries.
3. Idea Generation Layer: Employs the multi-model LLM pipeline to propose ranked prototype ideas annotated with complexity, estimated lines of code, and required dependencies.
4. 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.
5. 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.

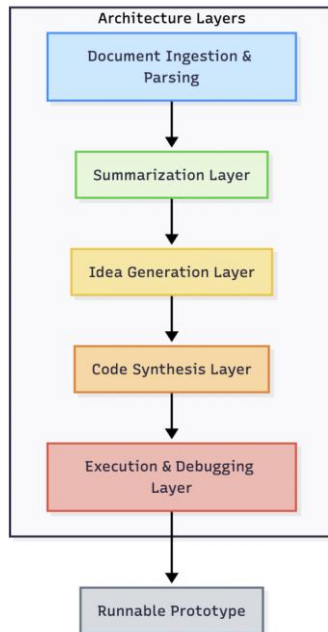


Fig. 1 System Architecture of PROTOGEN

V. ALGORITHMIC DESIGN

A. Algorithm 1: Paper-to-Prototype Pipeline

The paper-to-prototype pipeline operates as follows: (1) Input a 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 and TextRank hybrid; (5) Generate prototype ideas via NVIDIA NIM; (6) Select the most relevant idea by user or automatic ranking; (7) Use NVIDIA NIM to generate code for the selected idea; (8) Build a Docker image with an inferred dependency set; (9) Execute the code within the container. If execution fails, capture error logs, re-prompt the LLM (with Groq fallback if NVIDIA NIM is unavailable), rebuild and re-execute. Finally, expose the container port via an ngrok tunnel and return the live prototype URL.

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 evaluation.

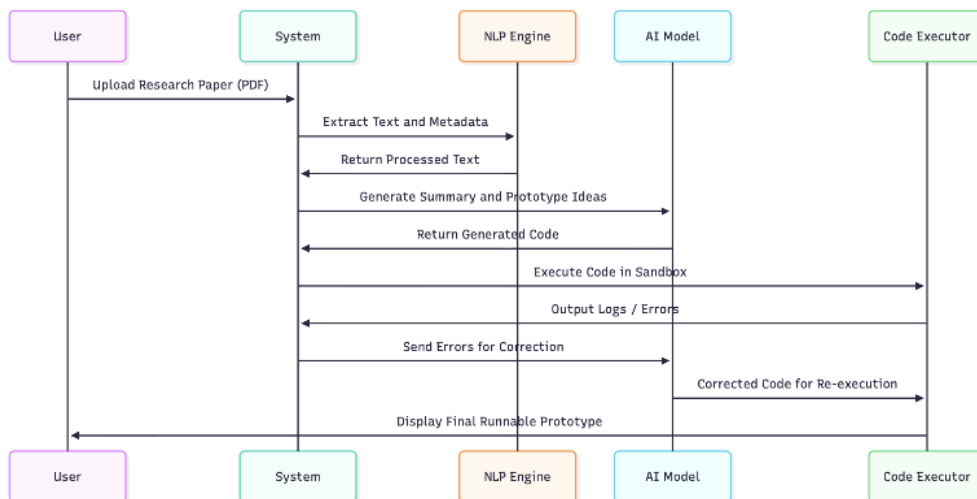


Fig. 2 Workflow Pipeline of PROTOGEN



VI. IMPLEMENTATION

A. Technology Stack

The PROTOGEN system is built on the following technology stack:

1. Frontend: ReactJS with Monaco Editor for code editing, Firebase Hosting for deployment.
2. Backend: FastAPI (Python) with asynchronous task handling.
3. Database: Firebase Firestore for project storage and chat history.
4. 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).
5. Execution: Docker for containerized code runtime; ngrok for live preview tunneling.
6. Document Processing: PyMuPDF and Tesseract OCR.

B. Docker Execution Pipeline

When the LLM generates a code prototype, PROTOGEN automatically: (1) parses import statements to infer a requirements.txt; (2) selects the appropriate base Docker image (e.g., python:3.11-slim or node:20-alpine); (3) builds a disposable container image with the generated code and dependencies; (4) executes the container and captures output streams in real time; and (5) exposes running web apps via an ngrok public URL, enabling live browser preview without user-side configuration. 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 adding a bar chart visualization to an existing model without restarting the full pipeline from scratch.



Fig. 3 High-Level Flow: Paper to Live Containerized Prototype

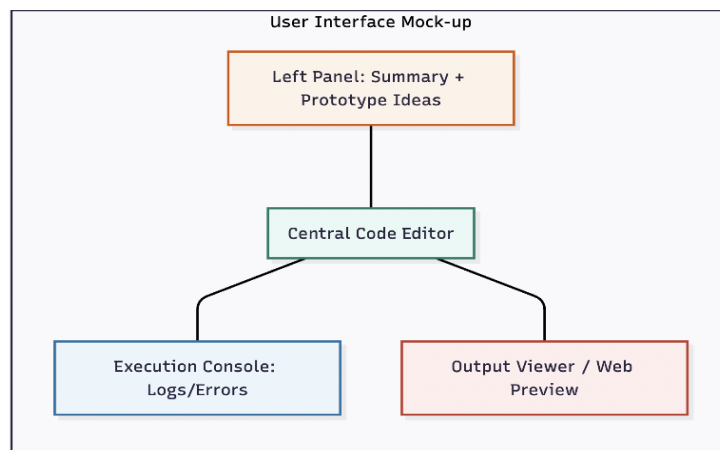


Fig. 4 User Interface Layout of PROTOGEN

VII. EVALUATION

A. Experimental Setup

PROTOGEN was evaluated 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

1. Prototype Success Rate (PSR): Fraction of generated prototypes that execute without fatal errors.
2. Pipeline Latency: End-to-end time from paper upload to live prototype URL.



3. Code Quality Score (CQS): Evaluated by two human experts on a 5-point scale assessing correctness, relevance, and readability.
4. Repair Iterations: Average number of LLM repair loops required before successful execution.

C. Results

TABLE I PROTOTYPE SUCCESS RATE BY DOMAIN

Domain	Papers	PSR (%)
Machine Learning	12	83.3
NLP	10	80.0
Computer Vision	8	75.0
IoT	8	87.5
Distributed Systems	7	71.4
Robotics	5	60.0
Overall	50	78.4

TABLE II COMPARISON WITH BASELINE APPROACHES

System	PSR (%)	Latency (s)	CQS
Manual implementation	100	~3600+	4.8
GPT-4 Code Interpreter	61.2	38	3.6
Piston API sandbox	54.0	12	3.2
PROTOGEN (ours)	78.4	42	4.1

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 Microservices 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 Understanding 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 Language Models for Code Generation," ACM Computing Surveys, vol. 56, no. 2, pp. 1–38, 2024.
- [9]. Z. Li et al., "CodeAgent: A Multi-Agent System for Unit Test Generation," arXiv preprint arXiv:2403.04984, 2024.
- [10]. 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.