EP2C: Explainable Paper-to-Code

University of Maryland

CMSC473 - Capstone in Machine Learning

September 20th, 2025

Vraj Patel | Ayhan Mehdiyev | Colin McElroy | Saleem Salavudheen

Github: https://github.com/vupatel08/473-Capstone-Project-EP2C

1. Background Review

Reproducibility has become one of the biggest challenges in machine learning research. Every year, thousands of new ML papers are published, but many do not release complete code or provide enough technical details to rebuild their experiments. As a result, reproducing these results is slow, frustrating, and often impossible without guesswork.

This lack of reproducibility is a problem because it slows down research progress, makes it harder for students to learn from real examples, and creates a gap between what is described in papers and what can actually be used in practice. Current tools that try to solve this problem only check if the code runs and do not explain why the code looks the way it does, what information is missing, or how specific code relates to the paper. Because of this, even if code is generated, researchers and students are left without guidance on how to fix errors or fully reproduce results. A system that can generate code and explain it would help close this gap and make ML research more accessible, transparent, and trustworthy.

The goal of EP2C is to generate both a repository and an explanation layer. This explanation layer links code files and functions back to sections of the paper, highlights missing datasets or hyperparameters, and generates a user guide with next steps. We will also define new evaluation metrics (static pass rate, guidance quality, and failure explainability) to measure the usefulness of these explanations. The project combines existing parsing tools (MinerU, Grobid), open-source code models (CodeLlama, StarCoder), and smaller fine-tuned models for explanation generation.

2. Related Work

AutoP2C introduced a multimodal, multi-agent system that extracts information from text, figures, and tables, then iteratively debugs code until it runs. Paper2Code, also known as PaperCoder, takes a top-down planning approach where LLMs first design a repository blueprint and then generate code to fill it. Both projects introduced benchmarks such as PaperBench and Paper2CodeBench to evaluate performance using replication scores and structural completeness. While these systems show that paper-to-code generation is possible, they primarily measure execution success and do not provide explanations or guidance for human users. The lack of explainability in paper-to-code is what our project is trying to address.

- ○ AutoP2C: https://arxiv.org/pdf/2504.20115
- ○ Paper2Code: https://arxiv.org/pdf/2504.17192

3. System Pipeline

Our EP2C pipeline ensures a comprehensive, user-friendly process for converting academic papers into reproducible, executable code with an additional layer of explainability. By combining parsing, code generation, debugging, and explainability into a seamless flow, EP2C provides a unique solution that not only makes research papers more reproducible, but also helps users understand and utilize the code.

a. Paper Parsing: Use MinerU? Gorbid?
   i. The user uploads the paper in their desired format (PDF, LaTeX, JSON, etc.)
   ii. Extract metadata such as the title, authors, abstract, DOI, keywords, etc.
   iii. Output structured metadata (most likely in JSON format) with textual content split by sections (abstract, introduction, methods, etc.)

b. Dataset/Code Search:
   i. Via HuggingFace's API, check if the paper already has a publicly available code repository and any datasets
      1. If no datasets are found, notify the user and provide steps on how they can obtain a dataset themselves
      2. If there is a dataset, use it / have it readily available for use within the code
      3. If there is a code repository already, the use it as a starting point and expand beyond with explanations

c. System Architecture Creation:
   i. The first parsing step was brief in order to get quick metadata to use in the second step, this parsing step goes much further in the breakdown and analysis of the provided paper
   ii. Go through all content in order to understand the structure of the proposed model/algorithm / whatever the paper proposes (such as an equation for math/physics papers)
   iii. Analyze any and all figures, diagrams, tables, etc., that need to be incorporated into the code
   iv. Ultimately output a blueprint for the code generation step

d. Code Generation:
   i. Feed the blueprint generated from the prior step into a fine-tuned LLM along with paper details to guide the code generation
   ii. The user is able to choose the programming language and set the complexity of the generated code (basic vs full-featured version) which will influence the level of detail in the generated code
   iii. The LLM will generate modular, dependency-aware code including code for dataset reading and handling/transformation (assuming a dataset is found or provided), code for defining the machine learning model, code for training loops, loss functions, evaluation and finally any unit tests or tests scripts that may be required.
      1. Note although this was talked about with ML papers in mind, this will include support for any type of paper within the field (such as mathematics or physics, which will obviously be less intense)
   iv. By the end of this step, we have an executable code repository in the user-choosen language

e. Iterative Evaluation (Debugging):
   i. Run basic tests on the generated code to check for any errors
   ii. Next, ensure proper code style/syntax via linting tools
   iii. Ensure all imports and external dependencies are correctly handled
      1. If the user has any missing/incorrect ones, provide detailed explanations on how to fix it
   iv. Iteratively fix any issues that come up, and rerun these tests after each fix to ensure no new bugs pop up and all code runs as expected
   v. By the end of this step, we have a fully working and debugged code repository

f. Into the UI we go:
   i. The UI will display the paper on one side and the generated code on the other (with one of the far sides having the file hierarchy) to allow users to see the code side-by-side with highlighted sections corresponding to parts of the paper

            1. Note: this needs to be fleshed out, but we might have to implement some sort of semantic matching algorithm to actually match corresponding parts of the paper to the code
            2. The goal is for when the user clicks on a code snippet the corresponding section in the paper will be highlighted (vice versa) showing the traceability

    g. Exporting:
        i. Users will have the option to download the generated code repository as a compressed .zip file
            1. This still needs to be checked out to see if it's worth, but users can also choose to authenticate themselves and push the generated code directly into their GitHub

4. Evaluation

EP2C will build on existing evaluation methodologies while also introducing new, explainability-focused benchmarks to assess the quality and usefulness of our auto-generated explanations. EP2C aims to offer a holistic approach to evaluating the usefulness of our auto-generated code repositories.

    a. Functional Evaluation Metrics:
        i. We plan to employ similar evaluation metrics used by AutoP2C and Paper2Code, to grade our own implementation
        ii. Replication Score:
            1. Measures how closely the generated code replicates the results within the paper
            2. Compare the experimental results obtained from the generated code with the original results in the paper
        iii. Structural Completeness:
            1. Measures how accurately the generated code follows the implementation presented in the paper
            2. Compare the structure of the generated code with the corresponding descriptions found in the paper
        iv. Error Rate:
            1. Track number of errors (syntax, logic, etc.) in the generated code
            2. Run simple tests to detect broken dependencies, logical errors, and syntax errors after the code generation.
    b. Explainability Evaluation Metrics:
        i. The critical difference between EP2C and existing models is the addition of an explanation layer. To evaluate this, we introduce new metrics to focus on the quality of said explanations
        ii. Static Pass Rate:
            1. Measures how often the generated code passes all tests without requiring any changes; our code implements an iterative evaluation phase at the end of the pipeline for debugging, so this metric tracks how often we get close to a perfect score without needing changes
            2. Run a test suite against our generated code (final step in the pipeline) and measure the percentage of tests that pass without needing a fix
        iii. Guide Quality:

1. Measures the clarity and usefulness of the generated explanations including comments, readme files, etc.
2. Given enough time, we can potentially assess this through human evaluation, but given the short amount of time we have, we will most likely use LLMs to assess the relevance of the generated explanations and how comprehensive they are
3. We want to score specifically on **clarity** (how well the code is explained), how **actionable** instructions are (if there are any needed), and on **completeness** to confirm all relevant details are covered such as dependencies and edge cases

iv. Alignment Accuracy:
1. Measures how accurately the generated code aligns with the paper's content
2. This mainly compares the code-paper traceability (the highlighted matching portions from code to paper)
   a. This is if we have enough time

5.

a. Choosing the Right LLM Model:
   i. CodeLlama has shown great promise for coding tasks (especially in python) and is highly optimized for this.
   ii. StarCoder is also designed specifically for code generation and works particularly well for tasks where high accuracy and domain-specific knowledge is important
   iii. GPT-4o is a strong general-purpose LLM

b. Fine-Tuning Strategy:
   i. Fine-tuning will mainly focus on papers that already have corresponding code (from HuggingFace, GitHub, etc.). The papers themselves will serve as the input text and the code they generate will be the target output.
   ii. We plan on using a supervised learning approach where the input is the academic paper (with sections like methods, models, algorithms, etc.) and the output will be the generated code repositories. Using instruction-based fine-tuning, we plan to make the model more adept at understanding complex algorithms and translating them into code.

c. Multiple Fine-Tuned Models:
   i. Given the numerous tasks we have within EP2C (code generation, explanation generation, debugging, etc), we may fine-tune separate models for each task
   ii. For instance we can fine-tune CodeLlama for our code generation, but then fine-tune a different model, like GPT-4o, that focuses on generating our human-readable explanations.

d. Creating the UI:
   i. Aside from all the API work, pipeline integration setup, fine-tuning, evaluating and testing our system, we still have to create the actual application to host and display our work.
   ii. This will need to be either a standalone application or a web-application, with considerations of how complex/simple we want it to be
      1. Mainly in regards to the tech stack, if we want to host this, if we want a database, etc.

   e. Third-Party API Integration:
      i. We need to utilize the HuggingFace API to search for relevant datasets and code related to the user-uploaded papers (as outlined in step 2 of the pipeline)
      ii. We might need to also work with the GitHub API if we want to allow users to export their code repositories into auto-generated repositories within their GitHub

6. Project Timeline
   a. Month 1: Initial Setup + MVP Development
      i. Divide work based on each member's strengths/interests
      ii. Gather academic papers for the dataset to fine-tune our LLM models
      iii. Begin fine-tuning LLM models
      iv. Set up the HuggingFace API as it's a crucial part of the pipeline
      v. Design the wireframe of the UI, quickly select a tech stack, and create the UI
   b. Month 2: Complete MVP + Testing/Evaluation
      i. Complete UI and integrate our pipeline within the backend of the UI
      ii. Run through several rounds of testing to ensure the app works as intended
         1. Given enough time, conduct UAT (user acceptance testing) and implement feedback
      iii. Begin evaluating and testing the system's performance based on our proposed benchmarks
   c. Remaining Time
      i. Conduct final rounds of testing
      ii. Address any bugs or issues
      iii. Refine UI for better usability and performance
      iv. Prepare case studies that showcase the effectiveness of the system in comparison to what's out there currently
      v. Write the final project documentation and prepare deliverables

7. Team Coordination
   a. Given this project is complex across multiple domains (machine learning, backend, frontend, etc.), we plan on placing communication on the forefront. We plan on trying our best to commit to weekly meetings to discuss progress, blockers, and align on next steps.
   b. We have also talked about using project management tools like Trello or organizing our tasks and assigning ourselves deadlines.
   c. We commit to using our group slack channel for any official forms of documentation sending/receiving, and our group chat within iMessages for quick messages
   d. Tasks at a high level:
      i. Gather academic papers with code for fine-tuning
      ii. Fine-tune LLM model for code generation
      iii. Fine-tune LLM model for explanation generation
      iv. Integrate HuggingFace API
      v. Design and develop frontend UI
      vi. Implement paper parsing and blueprint generation
      vii. Implement the iterative debugging/evaluation system
      viii. Implement code-paper matching
      ix. Build export functionality for downloading generated code as .zip files

        1. Potentially build out the GitHub repository exporting feature too, which is its own beast

    x. Integrate frontend with backend to display generated code and explanations in the UI
    xi. Conduct functional and usability testing
    xii. Create and manage custom evaluation metrics
    xiii. Develop case studies using real papers for evaluation
        1. If given enough time, conduct UAT and gather user feedback
    xiv. Write project documentation and final academic paper

8. Deliverables
    a. An EP2C web application or standalone application (depending on our final choice) that allows users to upload research papers, generated code, and view the code with explanations side by side
    b. Fine-tuned LLM models for code generation and explanation generation.
    c. A comprehensive evaluation report that includes the performance of EP2C based on the existing metrics and our new ones; this will include comparisons to the existing systems like AutoP2C and Paper2Code; this will also include our actual benchmark results
    d. An open-source GitHub repository containing the code for the entire project including who contributed to what
    e. A final academic paper outlining all our work including the system architecture, technical challenges, implementation details, a description of our pipeline and evaluation methodology, etc.