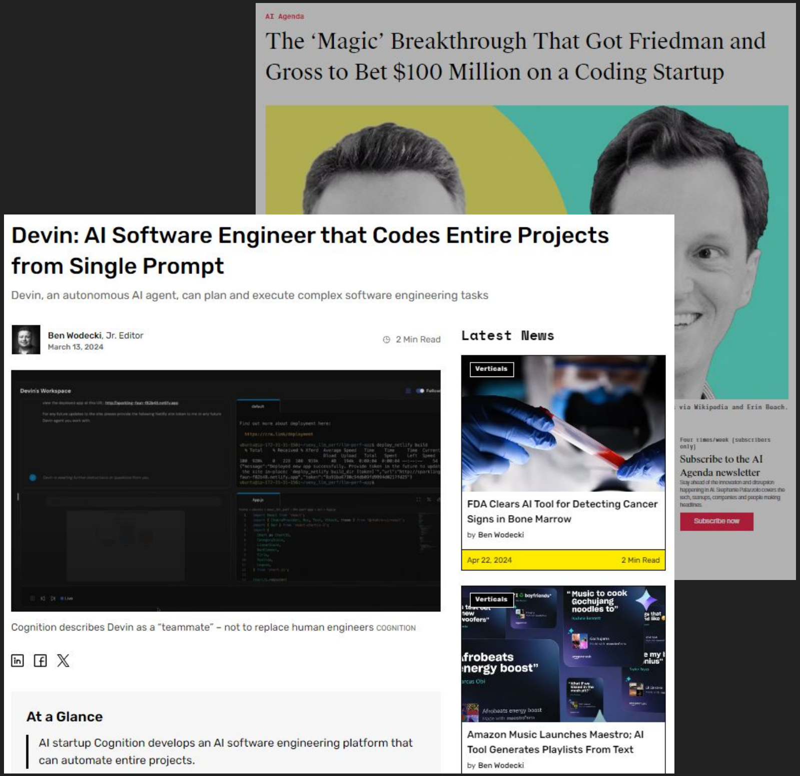
Agentic AI Workflows for DevOps

Asankhaya Sharma

When do you think AI will write 90% of all new code?

A. Less than 5 Years
B. More than 5 Years
C. Never

AI may automate software development tomorrow.



AI Agenda

The 'Magic' Breakthrough That Got Friedman and Gross to Bet $100 Million on a Coding Startup

via Wikipedia and Erin Beach.

Four times/week (subscribers only)

Subscribe to the AI Agenda newsletter

Subscribe now

Devin: AI Software Engineer that Codes Entire Projects from Single Prompt

Devin, an autonomous AI agent, can plan and execute complex software engineering tasks

Ben Wodecki, Jr. Editor
March 13, 2024

2 Min Read

Devin's Workspace

Cognition describes Devin as a "teammate" – not to replace human engineers COGNITION

At a Glance

AI startup Cognition develops an AI software engineering platform that can automate entire projects.

Latest News

Verticals

FDA Clears AI Tool for Detecting Cancer Signs in Bone Marrow

by Ben Wodecki

Apr 22, 2024 2 Min Read

Verticals

Amazon Music Launches Maestro; AI Tool Generates Playlists From Text

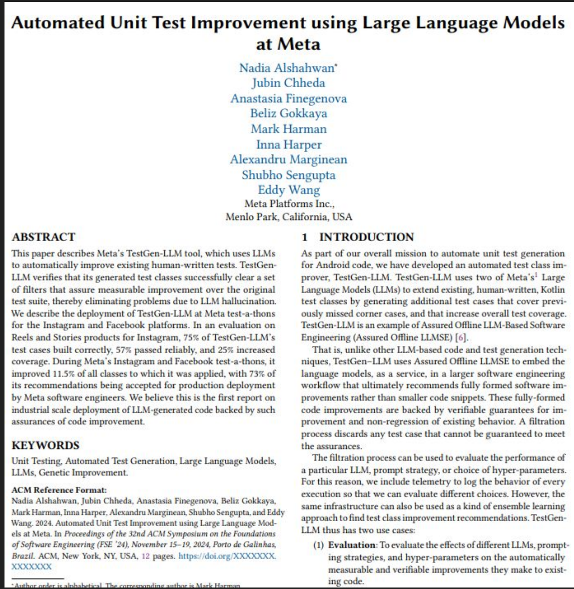
by Ben Wodecki

But we can automate the parts of development that suck, today.

Software Development Lifecycle today is built for human-written code.

It is laborious, disruptive and slow.

LLMs have a lot of potential to automate the SDLC...



Automated Unit Test Improvement using Large Language Models at Meta

Nadia Alshahwan*
Jubin Chheda
Anastasia Finegenova
Beliz Gokkaya
Mark Harman
Inna Harper
Alexandru Marginean
Shubho Sengupta
Eddy Wang
Meta Platforms Inc.,
Menlo Park, California, USA

ABSTRACT

This paper describes Meta's TestGen-LLM tool, which uses LLMs to automatically improve existing human-written tests. TestGen-LLM verifies that its generated test classes successfully clear a set of filters that assure measurable improvement over the original test suite, thereby eliminating problems due to LLM hallucination. We describe the deployment of TestGen-LLM at Meta test-a-thons for the Instagram and Facebook platforms. In an evaluation on Reels and Stories products for Instagram, 75% of TestGen-LLM's test cases built correctly, 57% passed reliably, and 25% increased coverage. During Meta's Instagram and Facebook test-a-thons, it improved 11.5% of all classes to which it was applied, with 73% of its recommendations being accepted for production deployment by Meta software engineers. We believe this is the first report on industrial scale deployment of LLM-generated code backed by such assurances of code improvement.

KEYWORDS

Unit Testing, Automated Test Generation, Large Language Models, LLMs, Genetic Improvement.

ACM Reference Format:
Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24), November 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As part of our overall mission to automate unit test generation for Android code, we have developed an automated test class improver, TestGen-LLM. TestGen-LLM uses two of Meta's[1] Large Language Models (LLMs) to extend existing, human-written, Kotlin test classes by generating additional test cases that cover previously missed corner cases, and that increase overall test coverage. TestGen-LLM is an example of Assured Offline LLM-Based Software Engineering (Assured Offline LLMSE) [6].

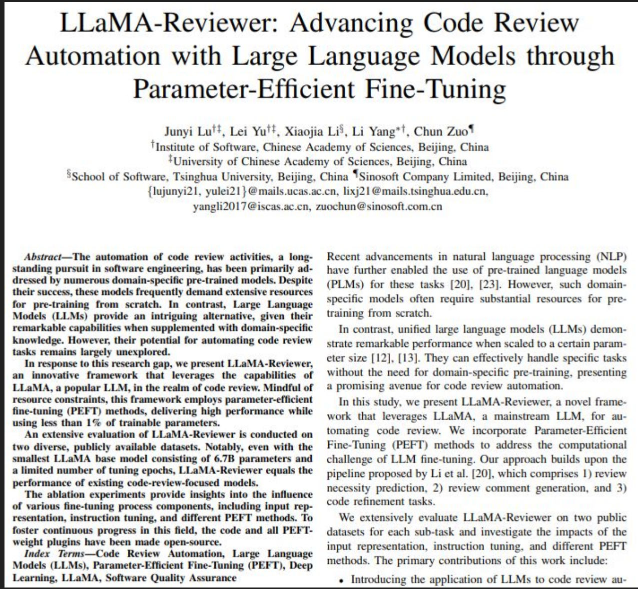
That is, unlike other LLM-based code and test generation techniques, TestGen-LLM uses Assured Offline LLMSE to embed the language models, as a service, in a larger software engineering workflow that ultimately recommends fully formed software improvements rather than smaller code snippets. These fully-formed code improvements are backed by verifiable guarantees for improvement and non-regression of existing behavior. A filtration process discards any test case that cannot be guaranteed to meet the assurances.

The filtration process can be used to evaluate the performance of a particular LLM, prompt strategy, or choice of hyper-parameters. For this reason, we include telemetry to log the behavior of every execution so that we can evaluate different choices. However, the same infrastructure can also be used as a kind of ensemble learning approach to find test class improvement recommendations. TestGen-LLM thus has two use cases:

(1) **Evaluation**: To evaluate the effects of different LLMs, prompting strategies, and hyper-parameters on the automatically measurable and verifiable improvements they make to existing code.

*Author order is alphabetical. The corresponding author is Mark Harman.

Unit Test Generation



LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning

Junyi Lu[†‡], Lei Yu[†‡], Xiaojia Li[§], Li Yang[*†], Chun Zuo[¶]
[†]Institute of Software, Chinese Academy of Sciences, Beijing, China
[‡]University of Chinese Academy of Sciences, Beijing, China
[§]School of Software, Tsinghua University, Beijing, China [¶]Sinosoft Company Limited, Beijing, China
{lujunyi21, yulei21}@mails.ucas.ac.cn, lixj21@mails.tsinghua.edu.cn,
yangli2017@iscas.ac.cn, zuochun@sinosoft.com.cn

***Abstract*—The automation of code review activities, a long-standing pursuit in software engineering, has been primarily addressed by numerous domain-specific pre-trained models. Despite their success, these models frequently demand extensive resources for pre-training from scratch. In contrast, Large Language Models (LLMs) provide an intriguing alternative, given their remarkable capabilities when supplemented with domain-specific knowledge. However, their potential for automating code review tasks remains largely unexplored.**

In response to this research gap, we present LLaMA-Reviewer, an innovative framework that leverages the capabilities of LLaMA, a popular LLM, in the realm of code review. Mindful of resource constraints, this framework employs parameter-efficient fine-tuning (PEFT) methods, delivering high performance while using less than 1% of trainable parameters.

An extensive evaluation of LLaMA-Reviewer is conducted on two diverse, publicly available datasets. Notably, even with the smallest LLaMA base model consisting of 6.7B parameters and a limited number of tuning epochs, LLaMA-Reviewer equals the performance of existing code-review-focused models.

The ablation experiments provide insights into the influence of various fine-tuning process components, including input representation, instruction tuning, and different PEFT methods. To foster continuous progress in this field, the code and all PEFT-weight plugins have been made open-source.

***Index Terms*—Code Review Automation, Large Language Models (LLMs), Parameter-Efficient Fine-Tuning (PEFT), Deep Learning, LLaMA, Software Quality Assurance**

Recent advancements in natural language processing (NLP) have further enabled the use of pre-trained language models (PLMs) for these tasks [20], [23]. However, such domain-specific models often require substantial resources for pre-training from scratch.

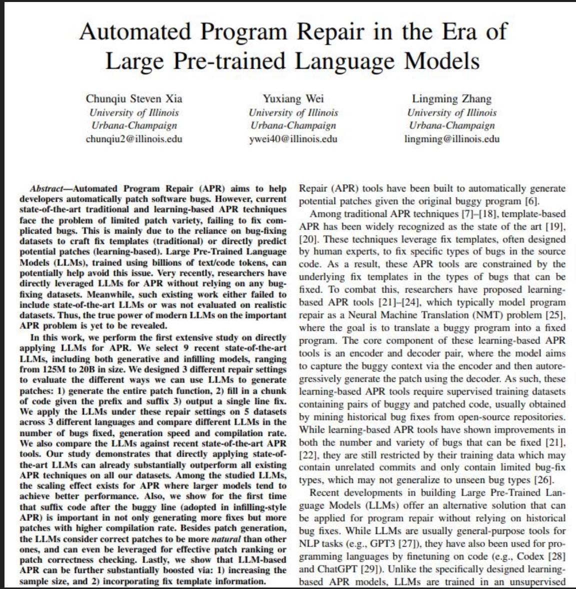
In contrast, unified large language models (LLMs) demonstrate remarkable performance when scaled to a certain parameter size [12], [13]. They can effectively handle specific tasks without the need for domain-specific pre-training, presenting a promising avenue for code review automation.

In this study, we present LLaMA-Reviewer, a novel framework that leverages LLaMA, a mainstream LLM, for automating code review. We incorporate Parameter-Efficient Fine-Tuning (PEFT) methods to address the computational challenge of LLM fine-tuning. Our approach builds upon the pipeline proposed by Li et al. [20], which comprises 1) review necessity prediction, 2) review comment generation, and 3) code refinement tasks.

We extensively evaluate LLaMA-Reviewer on two public datasets for each sub-task and investigate the impacts of the input representation, instruction tuning, and different PEFT methods. The primary contributions of this work include:

- Introducing the application of LLMs to code review au-

Code Review



Automated Program Repair in the Era of Large Pre-trained Language Models

Chunqiu Steven Xia, *University of Illinois Urbana-Champaign*, chunqiu2@illinois.edu
Yuxiang Wei, *University of Illinois Urbana-Champaign*, ywei40@illinois.edu
Lingming Zhang, *University of Illinois Urbana-Champaign*, lingming@illinois.edu

***Abstract*—Automated Program Repair (APR) aims to help developers automatically patch software bugs. However, current state-of-the-art traditional and learning-based APR techniques face the problem of limited patch variety, failing to fix complicated bugs. This is mainly due to the reliance on bug-fixing datasets to craft fix templates (traditional) or directly predict potential patches (learning-based). Large Pre-Trained Language Models (LLMs), trained using billions of text/code tokens, can potentially help avoid this issue. Very recently, researchers have directly leveraged LLMs for APR without relying on any bug-fixing datasets. Meanwhile, such existing work either failed to include state-of-the-art LLMs or was not evaluated on realistic datasets. Thus, the true power of modern LLMs on the important APR problem is yet to be revealed.**

In this work, we perform the first extensive study on directly applying LLMs for APR. We select 9 recent state-of-the-art LLMs, including both generative and infilling models, ranging from 125M to 20B in size. We designed 3 different repair settings to evaluate the different ways we can use LLMs to generate patches: 1) generate the entire patch function, 2) fill in a chunk of code given the prefix and suffix 3) output a single line fix. We apply the LLMs under these repair settings on 5 datasets across 3 different languages and compare different LLMs in the number of bugs fixed, generation speed and compilation rate. We also compare the LLMs against recent state-of-the-art APR tools. Our study demonstrates that directly applying state-of-the-art LLMs can already substantially outperform all existing APR techniques on all our datasets. Among the studied LLMs, the scaling effect exists for APR where larger models tend to achieve better performance. Also, we show for the first time that suffix code after the buggy line (adopted in infilling-style APR) is important in not only generating more fixes but more patches with higher compilation rate. Besides patch generation, the LLMs consider correct patches to be more *natural* than other ones, and can even be leveraged for effective patch ranking or patch correctness checking. Lastly, we show that LLM-based APR can be further substantially boosted via: 1) increasing the sample size, and 2) incorporating fix template information.

Repair (APR) tools have been built to automatically generate potential patches given the original buggy program [6].

Among traditional APR techniques [7]–[18], template-based APR has been widely recognized as the state of the art [19], [20]. These techniques leverage fix templates, often designed by human experts, to fix specific types of bugs in the source code. As a result, these APR tools are constrained by the underlying fix templates in the types of bugs that can be fixed. To combat this, researchers have proposed learning-based APR tools [21]–[24], which typically model program repair as a Neural Machine Translation (NMT) problem [25], where the goal is to translate a buggy program into a fixed program. The core component of these learning-based APR tools is an encoder and decoder pair, where the model aims to capture the buggy context via the encoder and then autoregressively generate the patch using the decoder. As such, these learning-based APR tools require supervised training datasets containing pairs of buggy and patched code, usually obtained by mining historical bug fixes from open-source repositories. While learning-based APR tools have shown improvements in both the number and variety of bugs that can be fixed [21], [22], they are still restricted by their training data which may contain unrelated commits and only contain limited bug-fix types, which may not generalize to unseen bug types [26].

Recent developments in building Large Pre-Trained Language Models (LLMs) offer an alternative solution that can be applied for program repair without relying on historical bug fixes. While LLMs are usually general-purpose tools for NLP tasks (e.g., GPT3 [27]), they have also been used for programming languages by finetuning on code (e.g., Codex [28] and ChatGPT [29]). Unlike the specifically designed learning-based APR models, LLMs are trained in an unsupervised

Bug Fixing

But they need to be integrated seamlessly,

with deep context,

and developer insights.

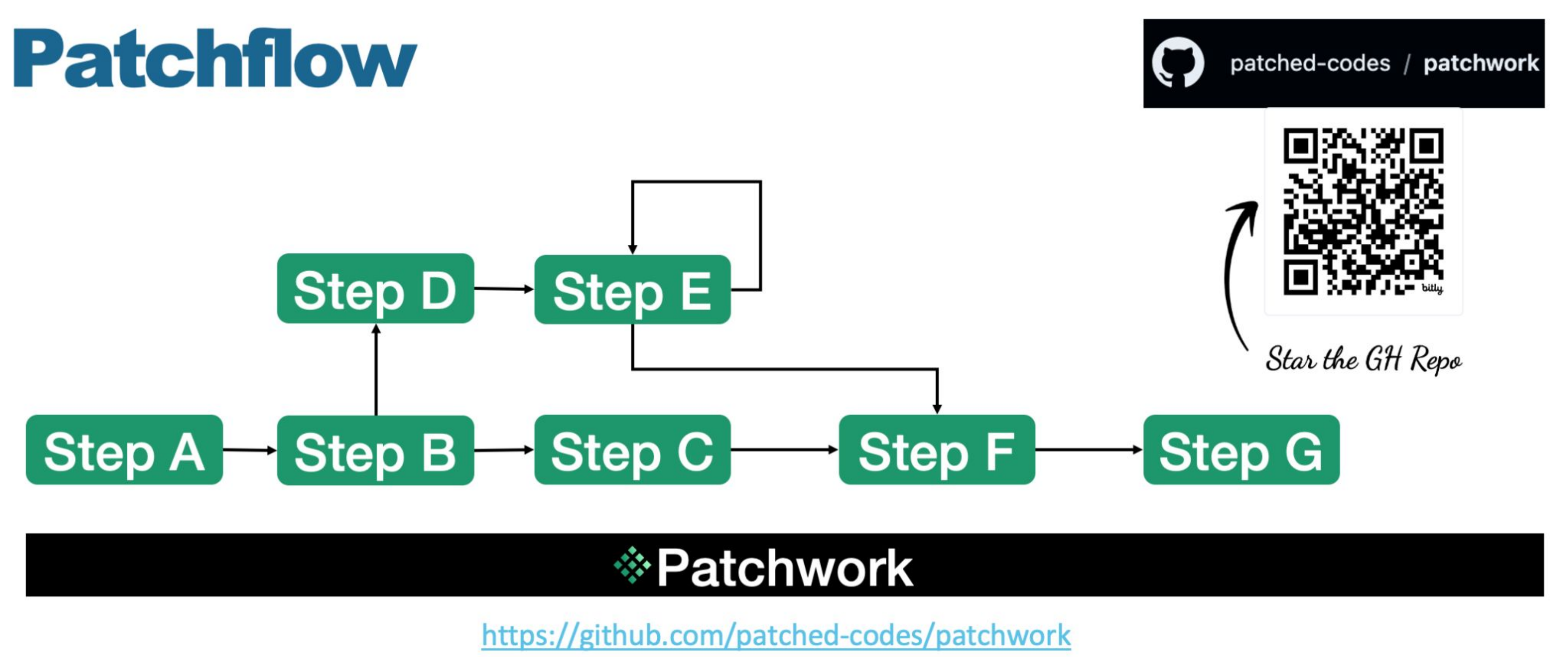
While preserving privacy,

and offering flexibility.

Introducing Patchwork

An open-source framework that effortlessly integrates into and automates your SDLC tasks, while giving you complete flexibility and control.

Solution



Patchflow
patched-codes / patchwork
Star the GH Repo
Step D
Step E
Step A
Step B
Step C
Step F
Step G
Patchwork
https://github.com/patched-codes/patchwork

Benefits

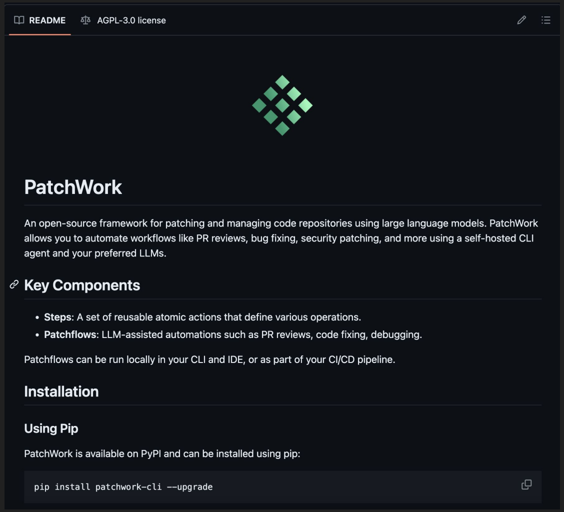
Integrated with IDE, CLI and CI

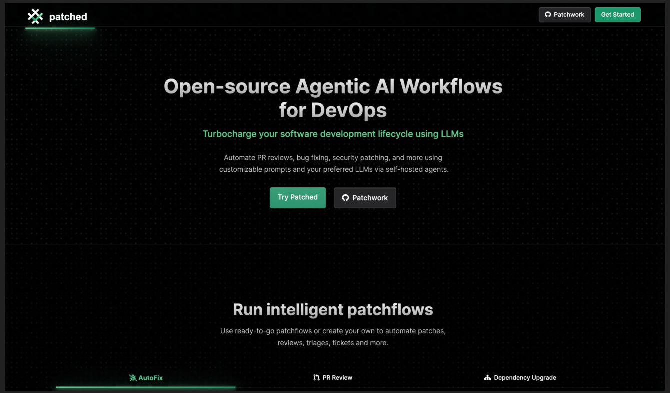
Customizable with prompt templates

Extensible with Steps

Works with any LLM

Demo





https://github.com/patched-codes/patchwork

https://patched.codes

Thank You!