Automated code reviews with LLMs

How Faire implemented an LLM-powered automated review pipeline to boost productivity

Luke Bjerring · Follow
Published in The Craft · 9 min read · 4 days ago

Large Language Models (LLMs) have made an enormous splash in the tech industry, improving various forms of natural language processing and task automation, especially with the launch of ChatGPT. Soon after OpenAI had launched its API offerings, Faire assembled an AI Foundations team in order to better understand and leverage this space. We ran a three-day AI Hackathon — much like our hack week — where contributors from across the company formed teams to prototype a wide range of LLM-powered feature ideas. The event was a raving success, with 92 projects submitted.

One of the projects where the Faire Engineering team ended up investing further was Automated Reviews. At Faire, we believe in the value of code reviews. The process is integral to how we write code. While there are many aspects of a code review that will require deeper context on a project, there are also a range of more generic review requirements that can be considered without the extra context. These include things like expecting a clear title and description, ample test coverage, styleguide enforcement, or spotting backward-incompatible changes across service boundaries.

Oftentimes, LLMs are well poised to perform the more generic aspects of the code reviews. With sufficient information about the pull request, such as the metadata, diff, build logs, or test coverage reports, LLMs can be prompted effectively to append useful information, flag dangerous changes, or even automatically fix simple issues.

In this article, we describe the review lifecycle and RAG (Retrieval Augmented Generation) setup that Faire has developed to power a range of context-specific automated reviews. We'll also include a spotlight focus on our test coverage review, which highlights areas with low coverage and suggests additional test cases, as a demonstration of the flexibility and value

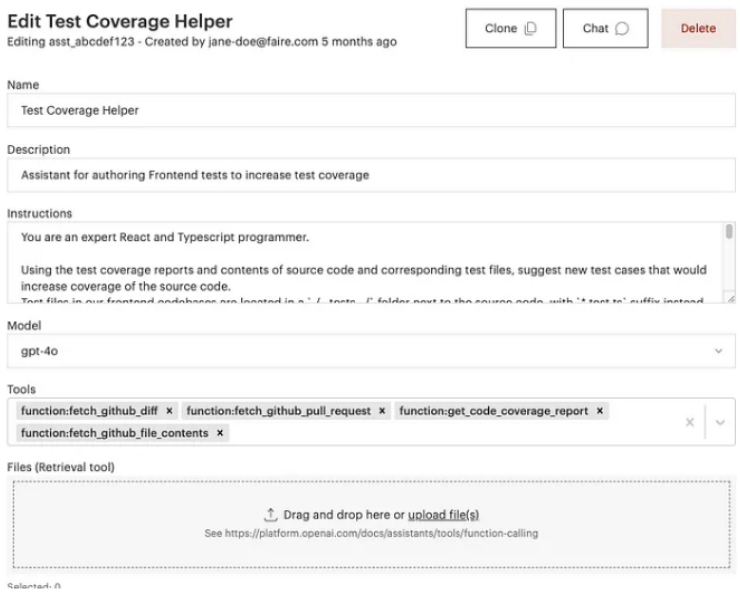
of review automation.

Fairey, our LLM Orchestrator service

Faire has developed an LLM orchestrator service, which we've named Fairey 🧚. This orchestrator service handles chat requests from our users, breaking them down into all the necessary steps that are required in order to produce a response. This includes calling an LLM model, fetching extra information, invoking functions, and other various logic.

Fairey is heavily integrated with OpenAI's Assistants APIs. We created a simple UI for managing AI assistants, such as tweaking instructions of the assistant or adding functions it can call. Functions give assistants additional capabilities, such as fetching additional information — a technique known as RAG (Retrieval Augmented Generation).

RAG is quickly emerging as an industry-standard approach to providing necessary information to LLMs in order to perform context-specific tasks. LLMs are trained on an extremely broad set of information, but typically not on your company's proprietary data. And, even if you've started fine-tuning open source models like Faire does, the LLM will usually still need any case-specific information — in our review pipeline, that's the information about the code changes being reviewed.



Screenshot of Faire's internal OpenAI Assistant manager.

Each function call has a callback defined in the orchestration service, which we invoke whenever GPT decides that we should. This allows Fairey to decide when to fetch the data that's needed, rather than having to send an overwhelming amount of extra information in the main prompt.

Review lifecycle

Fairey is also wired into Github's webhooks, which deliver event payloads whenever interesting things happen on pull requests. We react to each Github webhook in a variety of ways, including checking whether any automatic reviews are now ready to process the pull request. For pull requests that meet a review's criteria (e.g. programming language, or specific diff contents), Fairey will then interact with OpenAI to perform the reviews.

A simplification of how Fairey orchestrates interactions with Open AI's Assistant APIs.

After interacting with OpenAI, we check the output to see if Fairey has something useful to say. When it does, we emit the review on the pull request. Reviews usually include comments and hints, and may even include specific change suggestions for the code.

To avoid duplicate reviews, each review left by Fairey will also include some hidden metadata, so that we can check what's already been covered. This also potentially allows continuing upon previous review threads, or using the earlier review's outputs as inputs to incremental reviews.

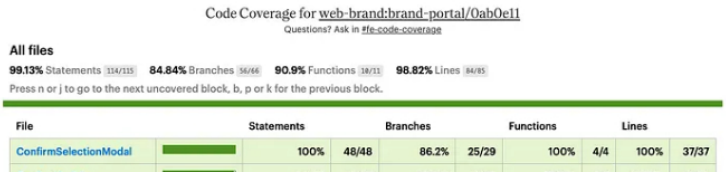
Test coverage review

To better demonstrate how Faire's automated reviews work, we'll take a deeper look at the test coverage review. Test coverage is a way of measuring whether particular branches of code are actually executed when running the test suite. For a simplified example, let's imagine a pull request that is implementing a new feature, guarded with a new setting from our Settings Framework. The code might look something like this:

```
const isSettingEnabled = getSettingValue();
if (myNewSetting) {
  doTheNewThing();
} else {
  doTheOldThing();
}
```

Test coverage tells us whether our tests cover the branches of the code where the setting is enabled and not enabled. Maybe we haven't added tests for the new code yet, and a code reviewer would point out that we need to!

In our frontend (web) codebases, Faire uses Jest for its unit test suites. We enforce that there is a minimum coverage for user-facing areas of code. Our test coverage in pull requests is computed incrementally, meaning when we create a pull request, our CI system will execute the test suite with the `--changedSince` flag, causing Jest to only execute tests that have a dependency on the source files included in the pull request. We also set the `--coverage` flag, which causes Jest to collect and output a coverage report, which can then be viewed in our internal development portal (see below for an example).



An example test coverage report for a pull request.

The test coverage review is triggered whenever we receive a webhook from Github that the `Test coverage` check run has completed with a `failure` outcome. When the coverage check has failed, it means the incremental coverage in the pull request is below our required coverage threshold. Usually, this will be because the author has added new source code files without any corresponding unit tests, or made new branches in the code that aren't covered by the existing tests.

The test coverage review uses a custom assistant (as seen in the screenshot above). It has access to several important function tools, including:

- **`fech_github_diff`** which grabs the diff of the changes in the pull request
- **`fetch_github_pull_request`** which gets the pull request's metadata such as title, description, etc.
- **`fetch_code_coverage_report`** which loads the lcov report, from our CI's build artifacts
- **`fetch_github_file_contents`** which loads the full contents of a given file

The assistant instructions, which are synonymous with a system prompt message, are something like:

```
You are an expert React and Typescript programmer.
Using the test coverage reports and contents of source code and corresponding
test files, suggest new test cases that would increase coverage of the source
code.

Test files in our frontend codebases are located in a `./__tests__/` folder
next to the source code, with `*.test.ts` suffix instead of `*.ts`. A couple
of examples,
```yaml
- source: `src/something.ts`
tests: `src/__tests__/something.test.ts`.
- source: `app/home/utils.ts`
tests: `app/home/__tests__/utils.test.ts`
```
```

Our codebases collect lcov code coverage reports on each pull request. The test coverage review is instructed to use the coverage reports to determine areas of the code with low coverage.

In addition to these system-level Assistant instructions, the review has a specific prompt template. The outgoing chat message will be something along the lines of:

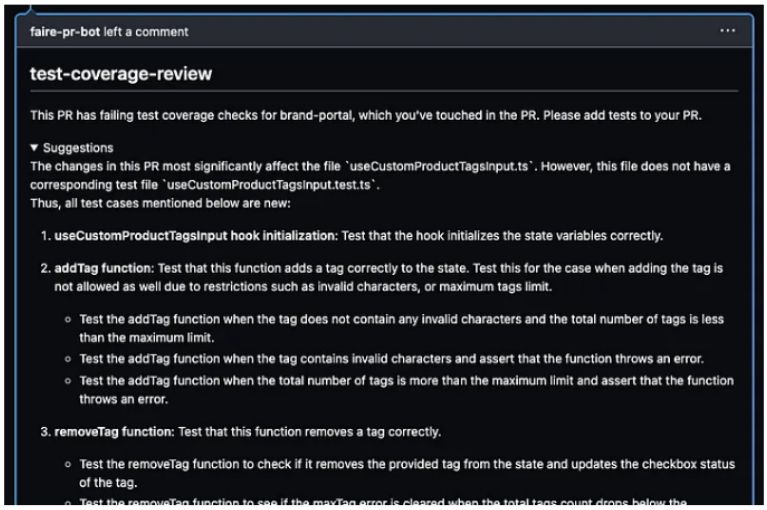
```
You are performing a code review of ${diffSource}. The PR has failing test cover

Use the diff and the code coverage report to identify the changes in the PR that
For those changes, suggest test cases that should be added to the code.
Use the existing test file as a reference when suggesting test cases. Do not sug
```

When a user puts up a pull request, our build pipeline will eventually report back with a failed `Test coverage` check run. This causes Fairey to kick off an

Assistant chat, using the prompt above. In order to produce a response, ChatGPT will issue function calls, which Fairey executes and submits back to ChatGPT, eventually producing a reply message.

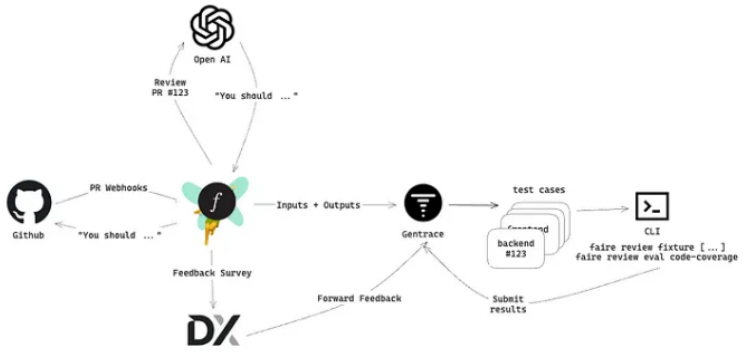
Then, we craft a Github Review using the reply, and post it to the pull request.



An example review for a pull request that didn't include any unit tests.

Evaluating the reviews

LLMs are incredibly flexible in their inputs, but just as varied in their output. GPT (generative pre-trained transformer) models are just predictive models, and they can make mistakes, outright hallucinate, and produce excessively verbose or unrelated output. We assess the quality of the reviews using two signals: one quantitative, and the other qualitative. Our quantitative evaluation involves using an LLM evaluation framework, while qualitative assessment involves surveying the end user (our engineers).



A simplified overview of how we assess the quality of the review outputs

In order to quantify the quality, we build out a set of test cases, iterate on the reviews, and re-run them, computing a correctness score. To achieve this flow, Faire makes use of LLM evaluation tools, such as those offered by Gentrace, CometLLM or Langsmith. Whenever we perform a review, we forward the inputs and outputs to the evaluation tools. Each input/output pair can be used to create a test-case. Then, as we iterate on the review (such as tweaking the prompt, changing the model, fetching different data, etc.), we can re-run the review pipeline across all of the test cases. Then, we use an LLM to assess the quality of the result.

That's right — the LLM review output is evaluated using...an LLM!

As for qualitative scoring, we leverage DX, a platform for measuring and improving developer productivity. DX is notified whenever a review is performed, and solicits survey responses from the author of the pull request. The feedback, good or bad, is then plumbed into Gentrace, where it can be used as a filtering signal for good test cases.

Fixtures

Pull requests reviews are quite transient in nature. Often, the information about the pull request relevant to the review will have changed by the time an engineer is ready to iterate on the behaviour and re-evaluate it.

To accommodate this, Fairey supports a "fixtures" feature, where the outputs of function calls are saved as snapshots for re-use in a later run. We achieve this using the OpenAI thread history, extracting the results of the function calls into fixture files which we upload to storage. When running a review at a later date, those fixture files are read and passed to Fairey as overrides for what ChatGPT should be given when it invokes a function.

Early impact of automated reviews

At Faire, we believe in empowering productivity. Automation of reviews using LLMs helps achieve this by streamlining the review process, reducing review latency for simpler problems, and freeing up our talent to focus on the most impactful and complex parts of the review — such as correctly meeting product requirements, implementing a clean architecture, long-term maintainability, and appropriate code reuse.

So far, we've seen success — as measured by positive user satisfaction and high accuracy — with automated reviews that:

- Enforce the style guide
- Assess the quality of the title and description
- Diagnose build failures, including issuing auto-fix actions
- Suggest additional test coverage
- Detect backward-incompatible changes

There is still room for growth — with each new type of review, the output quality will initially vary a lot. We've found that with LLMs, unwieldy input gives unpredictable output. Getting the trifecta of C's — consistent, concise, and correct — requires heavy refinement of the input content and structure, a broad set of test cases, and use of more complex prompting techniques such as self-eval and CoT (Chain-of-Thought). Using our iteration cycle, engineers are able to confidently iterate on the prompts and capabilities to drive efficacy.

When wielded correctly, LLMs can give engineers an exciting new level of assistance in rapidly delivering quality products to our users. We're excited to ride the wave of Generative AI.

. . .

This was a groundbreaking undertaking for Faire's Engineering team, but ultimately wouldn't have been pulled off without the tremendous contributions from Luke Bjerring, Btara Truhandarien, Justin Fangrad, Jordan Upiter, and of course, the team behind the original "Backward Compatibility Cop" AI hackathon prototype that inspired this project: Mike Welsh, Runqi Luo, Korosh Ahangar, Noah Eisen, James Oliver and Ximing Yang 🙏

. . .

Interested in joining a team doing interesting work that serves small businesses around the world? Apply for open roles: https://www.faire.com/careers

LIm ChatGPT Productivity Generative Ai Tools Engineering

158 1

Written by Luke Bjerring

28 Followers · Writer for The Craft

Senior Staff Engineer at Faire

Follow

More from Luke Bjerring and The Craft

Luke Bjerring in The Craft

Boosting performance: Faire's transition to NextJS

How Faire navigated the migration from a mature single-page application to the serve...

Jul 31 401 1

Faire Data Team in The Craft

Fine-tuning Llama3 to measure semantic relevance in search

Written by: Quentin Hsu, Wayne Zhang

Apr 29 241 3

Jason Xu in The Craft

Embedding-Based Retrieval: Our Journey and Learnings around...

Finding our ideal approach to search has

Luke Bjerring in The Craft

The benefits of auto-generated input forms with io-ts

At Faire, our internal Settings framework

required, well, a bit of searching.

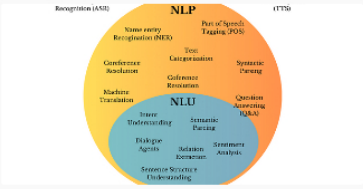
Jun 18 67 1

allows us to configure features and behavio...

Jun 29, 2021 73

See all from Luke Bjerring | See all from The Craft

Recommended from Medium



Vipra Singh

LLM Architectures Explained: NLP Fundamentals (Part 1)

Deep Dive into the architecture & building of real-world applications leveraging NLP...

Aug 15 1K 9

Thuwarakesh Murallie in Towards Data Science

How to Build Helpful RAGs with Query Routing.

An LLM can handle general routing. Semantic search can handle private data better. Whic...

Aug 16 272 1

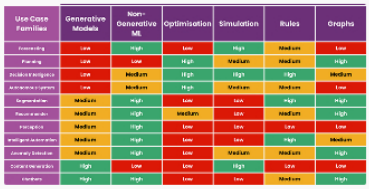
Lists

Our Favorite Productivity Advice
9 stories · 653 saves

Productivity 101
20 stories · 2240 saves

ChatGPT
21 stories · 768 saves

ChatGPT prompts
48 stories · 1923 saves



Christopher Tao in Towards AI

Do Not Use LLM or Generative AI For These Use Cases

Choose correct AI techniques for the right use case families

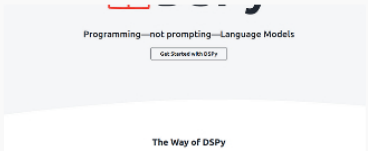
Aug 10 1.8K 22

Valentina Alto

Introducing Agent-based RAG

An implementation with LangGraph, Azure AI Search and Azure OpenAI GPT-4o

Aug 16 146 2



zhaozhiming in AI Advances

Advanced RAG Retrieval Strategies: Query Routing

An introduction to query routing in RAG retrieval strategies and how to use it during...

Aug 11 237

Vishal Rajput in AIGuys

Prompt Engineering Is Dead: DSPy Is New Paradigm For Prompting

DSPy Paradigm: Let's program—not prompt—LLMs

May 29 4.5K 46

See more recommendations