Resolving Code Review Comments with Machine Learning

Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, Petros Maniatis
Google

ABSTRACT

Code reviews are a critical part of the software development process, taking a significant amount of the code authors' and the code reviewers' time. As part of this process, the reviewer inspects the proposed code and asks the author for code changes through comments written in natural language. At Google, we see millions of reviewer comments per year, and authors require an average of ~60 minutes active shepherding time between sending changes for review and finally submitting the change. In our measurements, the required active work time that the code author must devote to address reviewer comments grows almost linearly with the number of comments. However, with machine learning (ML), we have an opportunity to automate and streamline the code-review process, e.g., by proposing code changes based on a comment's text.

We describe our application of recent advances in large sequence models in a real-world setting to automatically resolve code-review comments in the day-to-day development workflow at Google. We present the evolution of this feature from an asynchronous generation of suggested edits after the reviewer sends feedback, to an interactive experience that suggests code edits to the reviewer at review time. In deployment, code-change authors at Google address 7.5% of all reviewer comments by applying an ML-suggested edit. The impact of this will be to reduce the time spent on code reviews by hundreds of thousands of engineer hours annually at Google scale. Unsolicited, very positive feedback highlights that the impact of ML-suggested code edits increases Googlers' productivity and allows them to focus on more creative and complex tasks.

ACM Reference Format:
Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, Petros Maniatis. 2024. Resolving Code Review Comments with Machine Learning. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639477.3639746

1 INTRODUCTION

Google developers spend a substantial amount of time "shepherding" code changes through the code-review process: addressing simple review comments such as typo fixes and documentation expansion, asking for small refactoring to make the code more readable or efficient, or engaging in a dialog with the reviewer to resolve more involved and ambiguous feedback. Code review is primarily intended as a tool for high code quality, but it is also commonly used as an educational tool, as well as a knowledge-spreading tool [16, 19]. As such, it is highly regarded in the Google software-engineering culture as a valuable process to support, with both time and effort.

However, even in cases where only a single reviewing iteration is needed, there is a cost involved: the code author must understand the reviewer's recommendation, look up needed information such as APIs and local code conventions, and type out the edit; any interruption or "context switching" to another task dilutes the relevant context, and might require the code author to "page it all back in again" to complete the resolution of code comments in the future. It is therefore important to improve the ability of reviewers to give more actionable, precise suggestions, and to improve the ability of code authors to address such suggestions effectively. Addressing this need translates directly into increased productivity. Machine learning poses an emergent opportunity in this endeavor.

Much research effort has been expended recently on building machine learning models that "solve" software-engineering tasks, including code review (e.g., CodeReviewer [10]). Typically such results are predicated on the fast and impressive improvement of ML models of code [2, 4, 9, 18, 24], but are usually anchored on "lab" results: measurements of success metrics over evaluation benchmarks, with limited exposure to production use.

This paper describes our efforts bringing such research results to bear on an actual, rigorous, real-world code-review setting. We built an ML-based assistant that *resolves code-review comments* left by human reviewers. The specific assistance involves suggesting a code edit that addresses the reviewer's comment. We built, tuned, and deployed that assistant, and it has been in production use at Google for several months.

In this work, we side-step the question of the "best" ML architecture or foundation model, and turn instead to the problem of building an actual tool, used by thousands of Google engineers every day, and realizing its promise of strong impact on productivity.

Our contributions include:

- The careful curation of a training dataset drawn from tens of millions of code reviews that, joined with other software-engineering data collected at Google, led to an internal machine-learning model specializing in resolving code-review comments (§3).
- Lessons learned about tuning this model, associated datasets, and the resulting assistant design to improve prediction quality (§4).

ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0501-4/24/04.
https://doi.org/10.1145/3639477.3639746

- Lessons learned about the user-interface experience of the assistant that best supports our developers' productivity (§5).
- Qualitative and quantitative results about the positive impact of our deployed assistant (§6) in a real production environment, strongly demonstrating the feasibility and utility of the effort.

After several months of deployment, our code-review comment-resolution assistant is addressing roughly 7.5% of comments produced by code reviewers in their day-to-day work. Although we can share only some of the methodological details of our system, we hope that our experiences described here can inform other practitioners seeking to deploy ML assistance to real software-engineering practice.

The rest of the paper is organized as follows. §2 overviews the software-engineering landscape at Google where our assistant operates. §3 describes the design and implementation for the comment-resolution assistant. §4 presents our efforts toward higher model quality, and is followed by our efforts towards better usability in §5. §6 shares results from our initial deployment, and §7 outlines some recent related work at the intersection of code-review assistance with machine learning.

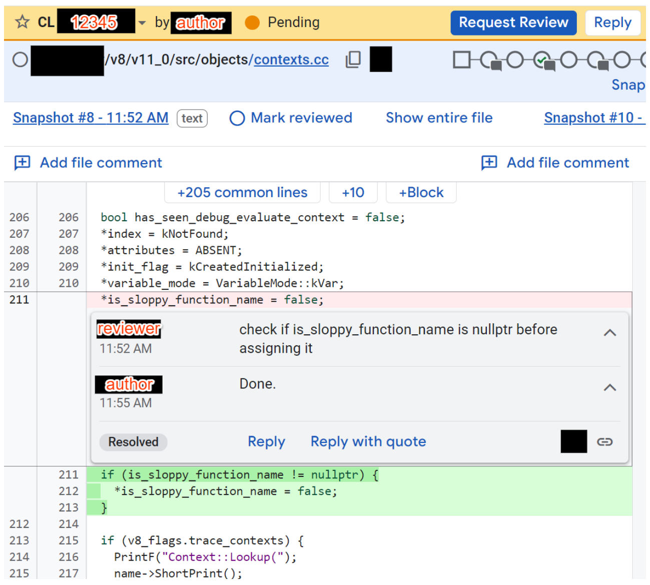
2 BACKGROUND: THE GOOGLE CODE-REVIEW WORKFLOW

We begin with a brief review of Google's software development process, especially with respect to code editing and code review, which are particularly relevant to this work.

Google stores its code in Piper [14], a monolithic source-code repository (also known as a *monorepo*). Code is updated via *changelists (CLs)* using Critique, Google's code-review system [25, Chapter 19]. A changelist is a transactional update of one or more files (including potential additions of new files or deletions of pre-existing files), corresponding to a particular purpose (described in the *changelist description*). A changelist is committed only after a review conducted by at least one peer, producing a new *revision* for each file included.

Code reviews involve design and functional vetting (i.e., does this changelist move the codebase towards the goals stated in some design document?), engineering checks (e.g., are the tests covering enough desirable and undesirable behaviors?), as well as a stylistic code-quality sign-off from a process called *readability review* [25, Chapter 3], whose purpose is to propagate best practices, share tips, and adhere to a consistent style guide [25, Chapter 8]. A consequence of this broad use of code review is that review comments can be as simple as "typo!" or as involved as a detailed question about the introduced change that requires complex program analysis to answer (e.g., "does this field ever become null?").

The developer addresses review comments by responding to them (e.g., to request clarification, or to argue that the comment is inapplicable), by editing the code to implement an explicit, uncontested suggestion (e.g., fixing the typo, and responding to the comment with "Done.") or even to implement an *implied* suggestion (e.g., responding to "does this field ever become null?" by checking if indeed this is possible, and if so, updating the code to check for that condition, which was the implied suggestion). After addressing comments in one review round, the code author creates a new



[redacted]/v8/v11_0/src/objects/contexts.cc [redacted]

```
bool has_seen_debug_evaluate_context = false;
*index = kNotFound;
*attributes = ABSENT;
*init_flag = kCreatedInitialized;
*variable_mode = VariableMode::kVar;
*is_sloppy_function_name = false;
```

reviewer 11:52 AM check if is_sloppy_function_name is nullptr before assigning it

author 11:55 AM Done.

Resolved Reply Reply with quote [redacted]

```
if (is_sloppy_function_name != nullptr) {
  *is_sloppy_function_name = false;
}

if (v8_flags.trace_contexts) {
  PrintF("Context::Lookup(");
  name->ShortPrint();
```

Figure 1: An example of a review comment in Critique. The reviewer asked for a defensive coding practice. The author addressed the comment by updating their changelist with a new review snapshot. The update is shown via colors: green for added text and red for removed. The author responded to the comment with "Done." and marked it "Resolved".

review snapshot and requests another look from the reviewer. See Figure 1 for an example.

If the reviewer is satisfied, marking all their prior comments as *resolved*, they declare the code review complete by responding with "LGTM" (i.e., "looks good to me"); if some old comments remain unresolved, or new comments pop up, the process continues, potentially in multiple rounds. Some changelists may be abandoned after zero or more review rounds, because the review process revealed the change was not advisable, or because they are replaced by other, perhaps smaller, changelists.

Editing is performed in copy-on-write workspaces in a system called Clients-in-the-Cloud (or *CitC* for short) [14]. During work in progress, changes to files from Piper are represented as successive *file snapshots*, which may be bundled as a changelist, reviewed, and possibly approved for inclusion to the Piper repository. CitC records all updates to files, even those that do not result in a changelist's review snapshots: every file save, including automatic editor saves—typically done every 30 seconds—produces a new CitC snapshot. Therefore, CitC snapshots provide a finer granularity of file evolution than review snapshots or Piper revisions.

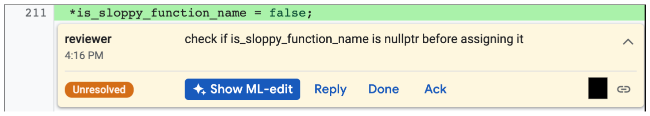
This workflow is similar to other source-control systems. For example, a changelist is akin to a GitHub pull request (PR), and review snapshots are reminiscent of git commits on a pull request.

3 LEARNING TO RESOLVE CODE-REVIEW COMMENTS

We now turn to our efforts using ML to produce a code-review assistant. §3.1 describes the vision guiding our path, §3.2 explains the ML architecture and training process for this assistant, and §3.3 describes how the assistant is deployed.

3.1 The Vision of ML-Assisted Comment Resolution

Given the existing code-review workflow (§2), we set out to build an ML-based software-engineering assistant that improves changelist velocity. Among the many opportunities for ML assistance, we focus on automated resolution of code-review comments. Specifically, we target an assistant that suggests an edit to resolve a given comment.



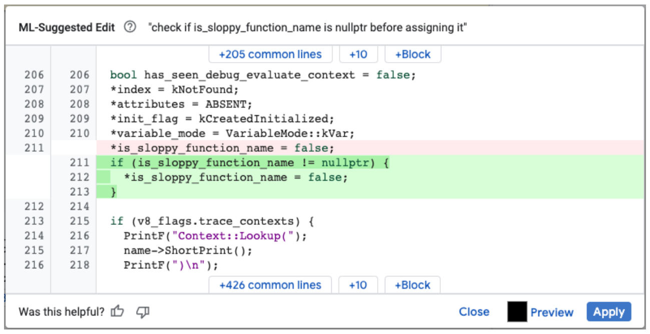
```
*is_sloppy_function_name = false;
```

reviewer
4:16 PM
check if is_sloppy_function_name is nullptr before assigning it

[redacted]

Figure 2: Critique notifies the author that there is an ML-suggested edit available.

Figure 2 shows an example, drawn from the same scenario as with Figure 1. The assistant suggests an edit to address the reviewer's comment and notifies the author. If the author decides to preview the suggested edit, they can click on "Show ML-edit", see the edit as a diff, and possibly apply that edit (Figure 3).



ML-Suggested Edit "check if is_sloppy_function_name is nullptr before assigning it"

```
bool has_seen_debug_evaluate_context = false;
*index = kNotFound;
*attributes = ABSENT;
*init_flag = kCreatedInitialized;
*variable_mode = VariableMode::kVar;
*is_sloppy_function_name = false;
if (is_sloppy_function_name != nullptr) {
  *is_sloppy_function_name = false;
}

if (v8_flags.trace_contexts) {
  PrintF("Context::Lookup(");
  name->ShortPrint();
  PrintF(")\n");
```

[redacted]

Figure 3: Critique shows the author an ML-suggested edit, which can then optionally be approved and instantly applied directly to the changelist.

In addition to choosing to apply an ML edit or to ignore it, the author also has the option to give specific feedback on the prediction with the thumbs-up/thumbs-down buttons. Applying an edit and clicking thumbs up for it are not equivalent; in some cases, a suggested edit may inspire a fix, but not be directly applied (but the author may click the thumbs-up button for it). Similarly, the thumbs-down button might indicate a more *intentional* negative feedback on the suggested edit, compared to clicking nothing at all (which might be a result of losing focus, switching to another task, and otherwise forgetting to click "Apply").

3.2 Modeling

We cast the problem of code-review comment resolution as a text-to-text machine-learning task. We target a text-to-text formulation,

Input Representation:

```
// [*] Task: Please fix the code-review comments.
  bool has_seen_debug_evaluate_context = false;
  *index = kNotFound;
  *attributes = ABSENT;
  *init_flag = kCreatedInitialized;
  *variable_mode = VariableMode::kVar;
* // [*] check if is_sloppy_function_name is nullptr before assigning it
  *is_sloppy_function_name = false;

  if (v8_flags.trace_contexts) {
    PrintF("Context::Lookup(");
    name->ShortPrint();
```

Example Edit Representation:

```
  bool has_seen_debug_evaluate_context = false;
  *index = kNotFound;
  *attributes = ABSENT;
  *init_flag = kCreatedInitialized;
  *variable_mode = VariableMode::kVar;
- *is_sloppy_function_name = false;
+ if (is_sloppy_function_name != nullptr) {
+   *is_sloppy_function_name = false;
+  }

  if (v8_flags.trace_contexts) {
    PrintF("Context::Lookup(");
    name->ShortPrint();
```

Figure 4: Example representation of the review comment from Figure 1. At the top, the input representation of the review snapshot and the comment inlined. At the bottom, the target edit for this particular comment resolution.

using a traditional Transformer [23] architecture based on T5 [15] (using the T5X [17] framework).

More specifically, the input to this ML model is the review snapshot at which the reviewer attached one or more comments. We "inline" each comment, by disguising it as a line comment in code, following the conventions of the programming language in which the example is written (e.g., using // for C++ or # for Python). We insert this artificial line comment at the first line where the reviewer attached their review comment. We do not consider comments attached to an entire file (without specifying a location), or an entire changelist (without specifying a file). The predictive target of this model is the edit that addresses the comment; we use an edit representation akin to diff patches. Figure 4 shows a training example drawn from Figure 1.

This representation can be applied to tasks other than review comment resolution, for example, addressing a compiler error, where instead of annotating the input with a comment text disguised as a line comment, we can annotate it with a compiler error (also disguised as a line comment). Similarly, the "inlining" of annotations as line comments can be used to *predict* such annotations in different tasks, e.g., predicting a likely static-analysis error by predicting the diff that would have inserted the annotation as a line comment. To differentiate among such tasks, we use a special code comment describing the task in natural language at the top of the example. In the figure, the task prompt is "Please fix the code-review comments." Finally, to fit a represented example within the model context window, we *prune* lines of the input, preferring to prune lines that are the farthest away from any inlined annotation.

In this fashion, we build a multi-task model for review comment resolution and other associated software-engineering tasks, using the DIDACT framework [12]. The tasks used to train this model, in addition to code-review comment resolution, include edit prediction (file snapshot to subsequent file snapshot in CitC), variable renaming, code-review comment prediction (predict the comment location and content from the review snapshot), build-error repair, etc., as well as the standard pre-training task for T5, *span denoising* (i.e., mask out a sequence of tokens and have the model predict them from context). The training corpus consists of over 3 billion examples, of which code review contributes about 60 million examples. We used a more "liberal" set of code-review examples during pre-training, including automated comments, and whole-file or whole-changelist comments. The samples used for fine-tuning and offline evaluation were solely human-generated line comments.

The model was trained using the standard cross-entropy loss, typical for such models, and tuned to maximize the exact sequence-match metric, that is, predicting the exact target for each example.

When used for inference, the model is tuned to attain a given desirable *precision* (i.e., number of correct predictions divided by all predictions). Specifically, every prediction of edits by the model is accompanied by a probability of that prediction; higher probability indicates higher confidence by the model in its prediction. To tune the model, we evaluate its predictions over a held-out *validation* dataset (say a few months of held-out code reviews not used during training). We choose a probability threshold such that the predictions with probability above that threshold achieve some desirable precision, e.g., 70%. The more aggressive this threshold is (i.e., the higher the target predictive precision), the fewer predictions are above threshold and, subsequently, the fewer predictions are shown to users. For some target precision *X*, we measure the *recall@X* of the model, by counting the number of correct predictions divided by all examples.

3.3 From ML Model to a Comment-Resolution Assistant

We focused on the overall user experience and developer efficiency in prototyping and deploying an assistant based on the model [5]. We explored different user-experience alternatives through a series of user studies and refined the feature assistant on insights from an internal beta (see §4 and §5). At a high level, we:

(1) Listen for incoming code-review comments produced by reviewers.
(2) Ignore comments produced by automated tools, comments not referring to a specific line, comments on unsupported file types, informational comments that the reviewer already marked as "Resolved", and comments that already have a suggested edit produced manually by the reviewer.
(3) Generate the model input in the same format as used for training (§3.2).
(4) Query the model and generate the suggested code edit.
(5) If the model is confident in the prediction (initially, targeting higher than 70% precision) and a few additional heuristics are satisfied, post the suggested edit to downstream systems.

To generate the code edit, each input is pre-processed into a textual model input using the source code, review comments, and other changelist metadata. The model then predicts the necessary changes to be applied to the input and rebased to the initial source code, perhaps resolving conflicts. The downstream systems, namely

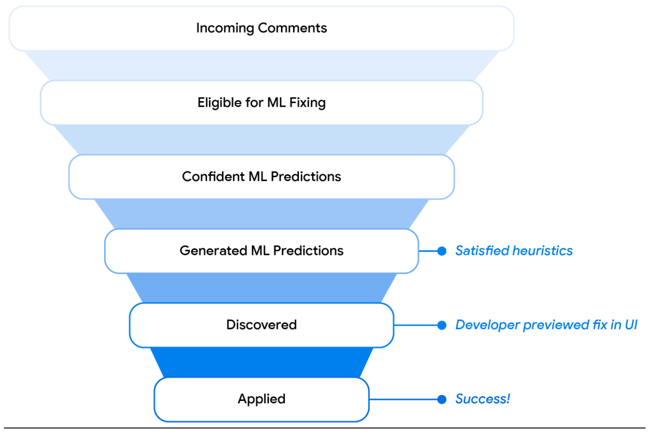


Figure 5: The journey of a comment to an ML-suggested edit applied to a code file.

the code-review frontend and the integrated development environment (IDE), expose the suggested edits to the user and log user interactions such as preview and apply events. Figure 5 illustrates the journey of a comment through the system. A dedicated pipeline collects these logs and generates aggregate insights.

4 THE QUEST FOR HIGHER MODEL QUALITY

The assistant produced in §3 went through a number of improvement iterations, with the goal to improve quality. Quality is measured in two ways. First, we use *offline evaluation*, by computing the recall@X metric described above over a held-out test dataset. Second, we use *user feedback*, by measuring the number of code-review comments produced during day-to-day business, the number of predictions the model made, the number of those predictions that were previewed, and of those how many were applied, or received thumbs up/thumbs down. All types of such evaluation are meant to detect an increase in developer productivity, but act as easier-to-measure proxies of that measure.

Offline evaluation is desirable because it does not inconvenience users, and can be done rapidly, with high frequency as the assistant is updated. On the other hand, offline evaluation is representative of a test dataset, which is drawn from code reviews that did not benefit from ML assistance, and were performed in an earlier time period (e.g., when reviewing guidelines may have been different); offline evaluation seeks to reproduce exactly what was observed. In contrast, online evaluation represents the current deployment environment, and involves the human user, collecting an assessment of the prediction, even if it is different from what a human developer might have produced. Out of all the user-feedback metrics, the most important is *acceptance rate*, the fraction of code-review comments that were addressed by an applied ML-suggested edit; other user-feedback metrics are typically used as measurable proxies for other qualities of the model's performance.

Refinements of the assistant were conducted first to improve offline evaluation performance. Then a beta tool was deployed to a small group of (friendly) users, and the assistant (both the model and the tool) was improved to increase both offline evaluation performance and user feedback metrics. Finally, the first milestone version, *V1*, was deployed to 50% of the Google population, and a careful A/B test was conducted to assess the user feedback on the resulting tool. After careful analysis, an improved milestone version, *V2*, followed the first deployment, went through an extensive beta phase, and is now deployed to the full 100% of the Google population.

We describe some dimensions of model-quality refinements below, and defer user-interface refinements for §5. Figure 6 illustrates improvements to recall@X metrics.

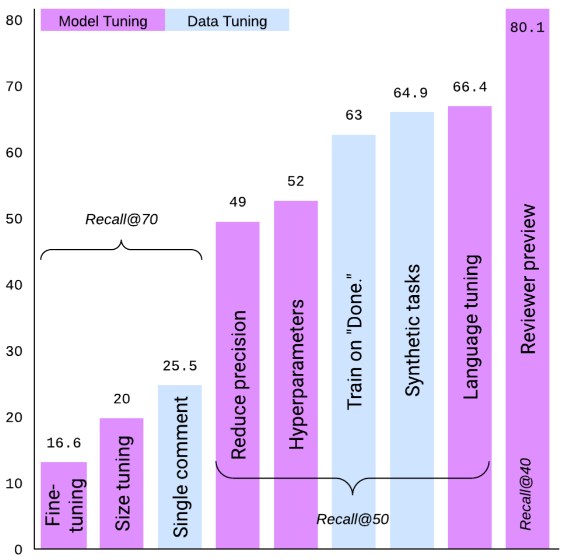


Figure 6: Illustration of model-quality improvements that affect recall@X.

Note that this is not meant to be a full ablation study of the different design decisions made at every evolutionary step, but merely a narrative of the various considerations in developing the assistant. A careful A/B study of each design update is, by necessity, time consuming—e.g., every effect must be isolated from other confounders, enough data volume is required to draw statistically significant conclusions, etc.—and was incompatible with our product-driven focus to deliver a useful assistant to our internal users in all due haste.

4.1 Model Tuning

Fine-tuning. The foundation DIDACT model we trained originally was targeting several software-engineering tasks. Fine-tuning that model on comment resolution exclusively (that is, further training on only comment-resolution examples) improved performance.

Size tuning. We experimented with various model sizes, trading off computational resources with model performance and latency. By increasing our model size (number of learnable parameters) by 2×, we improved recall by 25%. Further model-size increases did not improve offline performance.

Reduce precision. Initial feedback from the V1 beta users of the assistant indicated that a minimum precision of 70% was too conservative. Reducing the minimum precision to 50% increased the absolute recall value (which is unsurprising), without reducing user satisfaction (e.g., the rate of suggestions that were accepted, as well as thumbs-up ratings).

Hyperparameters. We carefully explored the hyperparameters of the fine-tuned model, further improving recall@50.

Language Tuning. We noticed that minimum-precision thresholds for different languages vary, leading to reduced performance when using a single threshold for all predictions. We introduced per-language minimum-precision threshold tuning, and that increased the aggregate recall@50 metric.

Reviewer Preview. When reviewers were given the ability to approve or reject suggested edits (§5), we were able to reduce the target precision further to 40%, which increased recall@40 even further, while still improving downstream acceptance.

4.2 Data Improvements

Single-comment. Reviewers often provide more than one comment during a review. Consequently training examples may include multiple comments, all of which are addressed in a single *entangled* edit. Initial prototype explorations with users showed that developers prefer an edit attributable to a single comment, rather than an undifferentiated set of edits addressing multiple comments at the same time. We therefore predict a single suggested edit for a comment, ignoring all other comments in the file. We found that reducing the training data to examples with a single review comment does not improve model quality. However, we found that limiting our offline evaluation dataset to single-comment edits represented our production scenario better, and led to more confident, higher-precision predictions.

Train on "Done." Some review comments are accepted and addressed immediately (and receive a formulaic "Done." response, produced with a single button click). Other comments may lead to a discussion, and in some cases rejection of the reviewer's suggestion. We found that by filtering the training examples to comments that are immediately accepted and have a single "Done." response, we improved model recall by more than 20%.

Synthetic tasks. The fine-tuned model sees comment-to-edit examples drawn from Google's history of code reviews. We further augment the dataset with some synthetic tasks that disguise other code-editing activities as code review, to provide the model with additional edit opportunities: 1) A code-infilling task, in which a function, a function body, a part of a function body, or a unit test are removed, and a code-review comment is introduced in their place prompting the model to implement the missing code, sometimes by reusing the pre-existing documentation string as part of that prompt. 2) A build-repair task, in which a file snapshot on which the compiler failed to compile, is cast as a code-review comment of the sort "Please fix the broken build below." There already is such a build-repair task in the DIDACT pre-training mixture, and this synthetic task casts the same examples in this alternative prompting format, and the target edit is the one that fixes the compiler error.

3) A set of comment resolutions in which the developer rewrote the predicted suggestion, collected from the production system. This is a form of incidental instruction tuning.

4.3 Prediction Post-Processing

We manually sampled suggested edits and used *thumbs-down* clicks from beta users to identify recurring failure patterns. We implemented serving-time heuristic filters to reduce the number of shown incorrect predictions. With these changes, we traded reduced quantity of shown suggested edits for higher quality of user experience and an increase in the observed acceptance rate in production. Specifically:

Filter out: Spread-out Edits. Filtering suggested edits that are more than 5 lines away from the comment reduces the rate of thumbs down clicks for previewed edits from 8.5% to 6.7%.

Filter out: Promises to do it Later. The model sometimes resolves a comment by adding a promise to do it later in a "to do" code comment! When the assistant suggested such edits, users marked them with a thumbs down, so we do not surface such predictions to users.

Filter out: Delete Code In Scope. The model addresses a few comments incorrectly by deleting large blocks of code where the reviewer's comment was attached. Although sometimes such massive deletions are indeed the intended resolution, we decided to drop all such delete-only predictions, to protect user trust.

4.4 A Curated Qualitative Evaluation Dataset

A valuable tool in improving assistant quality was a *manually-curated, qualitative evaluation dataset*, in addition to the existing, held-out quantitative validation and test data splits from the training dataset. Such curation is motivated by the need to track performance in example categories that are not frequent in our other training and evaluation datasets. We also targeted a small dataset size (on the order of 100 examples), so that a single person can inspect the full set of predictions and ground truth side-by-side, without fatigue, and in enough detail to rate predictions as correct, incorrect, or "close".

We curated examples with the following criteria:

- A few simple examples that allow us to detect regressions.
- A few difficult examples, already within the capabilities of the current model, also used to detect regressions.
- A few examples that require changes spread across multiple clusters within the file.
- A few examples that are sensitive to the file position where the review comment is attached.
- A balanced mix of examples that focus on code refactoring and code generation.
- A few examples that we *considered feasible*, but that the model failed to resolve.
- A few examples that we *considered feasible*, but that led the model to generate only deletions.
- A few "moonshot" examples that we considered infeasible for the model, to assess performance headroom and failure commonalities (e.g., to ensure that the model has low confidence for out-of-distribution comments).

We sourced these examples manually 1) from the held out splits (validation and test) of the training data, 2) from accepted suggested edits in production, 3) from provided manual rewrites by users, and 4) by explicitly generating examples that fulfill a desired property. We explored additional sources such as suggested edits that have been previewed by the user but not applied. We found these very noisy and not suitable for our goal.

Anecdotally, as the model quality improved, during evaluations on the curated datasets, we sometimes identified model suggestions that *improved upon our previous ground truth*! We often kept multiple correct alternatives, to simplify the job of the human rater.

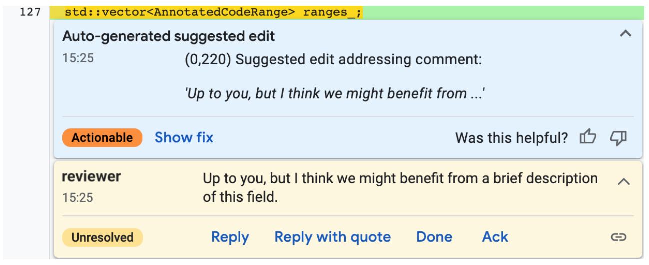
All in all, the curated dataset served as a qualitative, diverse regression test for desirable properties of the assistant, as a way to formalize launch blockers (e.g., delete-only edits) and updated priorities in the complexity of intended resolutions, and as a way to capture behaviors that did not have sufficient frequency in our training/evaluation data splits to appear naturally in our held-out test split, especially for cases where the distribution changed over time, altering what "ground truth" was.

By necessity, such a curated dataset has a limited useful lifetime and must be updated. For example, what is considered a "moonshot" becomes feasible when model quality improves dramatically.

5 THE QUEST FOR HIGHER USABILITY

Even a high-quality ML model for code-review comment resolution can be wasted if not presented to users in an effective, usable manner. We next describe the evolution of our assistant, and the pitfalls we uncovered that, when mitigated, improved its usability. As before, acceptance rate (i.e., the fraction of comments resolved by an accepted ML suggestion) is an important metric of usability, but a core auxiliary metric is also *discoverability*: the fraction of surfaced suggestions that were previewed by system users.

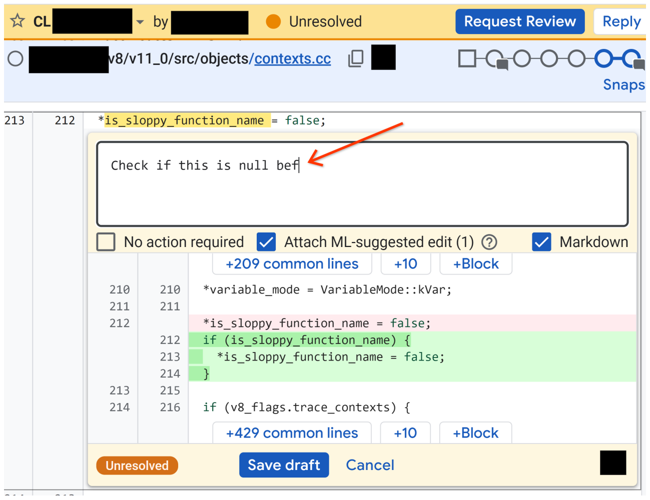
Note that sometimes usability gaps (e.g., low discoverability) might require modelling improvements (e.g., a different data representation), although this was not necessary in the versions of our assistant described in this article.


```
std::vector<AnnotatedCodeRange> ranges_;
```
Auto-generated suggested edit
15:25 (0,220) Suggested edit addressing comment:
'Up to you, but I think we might benefit from ...'

reviewer
15:25 Up to you, but I think we might benefit from a brief description of this field.

Figure 7: The initial Critique integration for the code author that shows *Auto-generated suggested edits* in a distinct annotation attached to the same text range where the reviewer comment was attached.

Pitfall: decoupled comment and suggested edit. Our first integration of the ML model in the Critique UI was by running a separate, asynchronous analyzer that queried the model and produced the suggested edit as an independent code finding, in a separate annotation (see Figure 7). This resulted in duplication of information, wasted precious UI real-estate, and confused the prevailing visual language of review comments. Our V1 beta evaluation showed that only 20% of suggested edits were previewed by developers through a click on the "Show fix" button. A UI update that combined the two sources of information, by placing a "Show ML edit" in the same box where the reviewer comment appears (Figure 2) improved discoverability considerably (up to about 30% at V1 launch).


☆ CL [redacted] by [redacted]
Unresolved
Request Review
Reply
[redacted] v8/v11_0/src/objects/contexts.cc [redacted]
Snaps

```
*is_sloppy_function_name = false;
```

```
Check if this is null bef
```

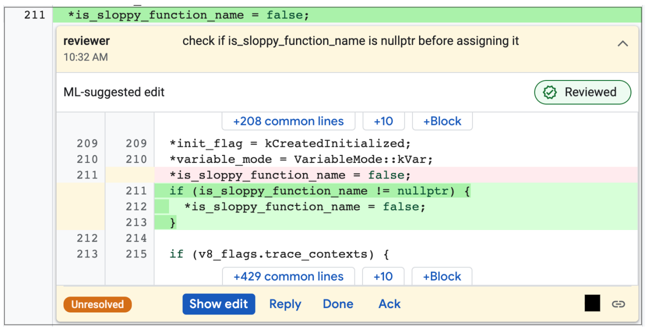
[ ] No action required [✓ Attach ML-suggested edit (1)] [✓ Markdown]

+209 common lines +10 +Block

```
*variable_mode = VariableMode::kVar;

*is_sloppy_function_name = false;
if (is_sloppy_function_name) {
  *is_sloppy_function_name = false;
}

if (v8_flags.trace_contexts) {
```

+429 common lines +10 +Block
Unresolved
Save draft
Cancel
[redacted]

Figure 8: As the reviewer started typing the comment, the suggestion was shown. Note that unlike the more descriptive text of Figure 2, the reviewer had not yet had a chance to specify the details (e.g., "before assigning") or even designating the particular variable in question. The location of the comment and the mention of "check" and "null" were sufficient to trigger the assistant to suggest the intended edit. It should be noted though that earlier, as the comment is being typed, the assistant will suggest a rename or other unrelated assignments. The reviewer has the option to continue typing until they complete their comment (and possibly discard the suggested edit if it is inapplicable), stop when they see what they intended, or complete and elucidate the comment even after the right suggestion is predicted by the assistant.



```
*is_sloppy_function_name = false;
```

reviewer
10:32 AM
check if is_sloppy_function_name is nullptr before assigning it

ML-suggested edit

```
*init_flag = kCreatedInitialized;
*variable_mode = VariableMode::kVar;
*is_sloppy_function_name = false;
if (is_sloppy_function_name != nullptr) {
  *is_sloppy_function_name = false;
}

if (v8_flags.trace_contexts) {
```

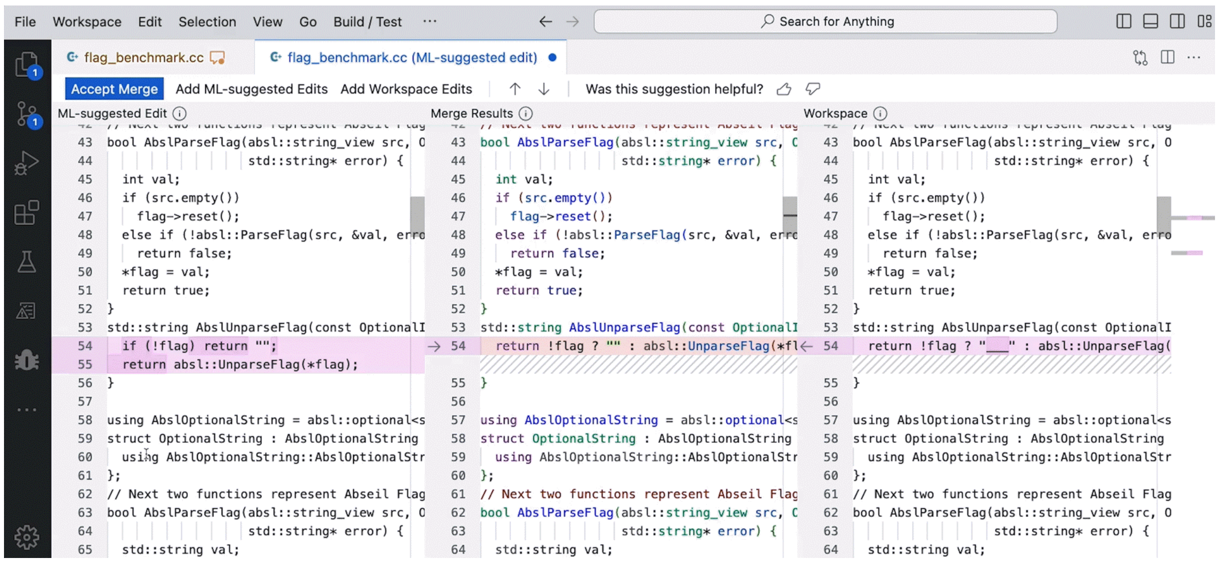
Figure 9: The author is shown an ML-suggested edit that the reviewer previewed and approved.

Pitfall: decoupled reviewer from ML assistant. The most frequent negative feedback during the beta of the V1 assistant was from reviewers who were uncomfortable having an ML model "interpret" their comment into a suggested edit, and would prefer to preview the suggestion before providing it to the code author; the pedagogical function of code review was of primary concern here, since code review is often how new engineers are trained on local conventions and programming discipline. This led to a UI revision in V2, in which the *reviewer* is shown the ML-suggested edit (Figure 8) as they type their comment. The reviewer can decide to reject the suggested edit (in which case the author will only see the reviewer's comment). If the reviewer takes no action, the author sees the comment along with the suggestion, as well as an additional annotation of the suggested edit denoting that is has been "Reviewed" (Figure 9). We noticed initially that reviewers only approved about half of the ML-suggested edits to their own comments, which suggests that the quality of suggested edits reaching code authors increased as a result of this pre-approval. Furthermore, since the reviewer can reject obviously incorrect suggested edits, we could be even more aggressive with ML confidence thresholds, dropping the V2 minimum precision to 40% from 50% and, therefore increasing the number of comments that received ML suggestions.

Pitfall: slow-to-predict edits. Reviewer-previewed suggested edits are valuable to code authors (see above); however, reviewers are typically pressed for time, and may move on quickly from comment to comment. In an attempt to reduce back-end prediction load, and to avoid showing reviewers suggested edits before they have typed enough of their comment, we set the triggering delay between when the reviewer starts typing a comment and when a prediction is requested to 1500ms. Between this triggering delay, and the additive prediction latency of the model, many predictions "missed" the reviewer, who had already moved on. When we further reduced the triggering delay to 500ms, and improved the prediction latency through considerable engineering effort, the number of suggested edits previewed by reviewers increased by 12%, and the acceptance rate of ML-suggested edits by authors improved by 18%. We found the 2x increase in back-end load that resulted from this triggering change well-justified by the improvement in acceptance rates.

Pitfall: click to view. Since code *shepherding* (i.e., editing the changelist in light of the reviewer comments) takes a significant fraction of developers' time—one study at Google measured the median to be around 60 minutes [7]—efficiency in addressing comments is important. We found that ML-suggested edit discoverability for the changelist author improved when we started showing the suggested edit immediately next to the reviewer comment (Figure 9), rather than requiring a click of the "Show ML edit" button (Figure 2).

Pitfall: code review is serialized. Our original design of the UI assumed that the changelist author and reviewers operate in lock step: one stops when the other starts working on the changelist. This is not, however, how code review operates in practice. Sometimes the changelist is edited by the author as the reviewer is reviewing, or perhaps the reviewer thinks of a new comment after they have passed the bulk of their review to the author, and sometimes the reviewer attaches a comment to an older review snapshot of



ML-suggested Edit

```
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  int val;
  if (src.empty())
    flag->reset();
  else if (!absl::ParseFlag(src, &val, erro
    return false;
  *flag = val;
  return true;
}
std::string AbslUnparseFlag(const OptionalI
  if (!flag) return "";
  return absl::UnparseFlag(*flag);
}

using AbslOptionalString = absl::optional<s
struct OptionalString : AbslOptionalString
  using AbslOptionalString::AbslOptionalStr
};
// Next two functions represent Abseil Flag
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  std::string val;
```

Merge Results

```
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  int val;
  if (src.empty())
    flag->reset();
  else if (!absl::ParseFlag(src, &val, erro
    return false;
  *flag = val;
  return true;
}
std::string AbslUnparseFlag(const OptionalI
  return !flag ? "" : absl::UnparseFlag(*fl
}

using AbslOptionalString = absl::optional<s
struct OptionalString : AbslOptionalString
  using AbslOptionalString::AbslOptionalStr
};
// Next two functions represent Abseil Flag
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  std::string val;
```

Workspace

```
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  int val;
  if (src.empty())
    flag->reset();
  else if (!absl::ParseFlag(src, &val, erro
    return false;
  *flag = val;
  return true;
}
std::string AbslUnparseFlag(const OptionalI
  return !flag ? "___" : absl::UnparseFlag(
}

using AbslOptionalString = absl::optional<s
struct OptionalString : AbslOptionalString
  using AbslOptionalString::AbslOptionalStr
};
// Next two functions represent Abseil Flag
bool AbslParseFlag(absl::string_view src, O
                   std::string* error) {
  std::string val;
```

Figure 10: Three-way merge window pops up in the IDE when the author attempts to accept an ML-suggested edit in an incompatible code state.

the changelist. All in all, this means that sometimes even an ML-suggested edit that the author wishes to accept is incompatible with the current state of the code. We found that by detecting those cases, and opening a three-way merge window (Figure 10) for the author to resolve any merge conflicts, the number of accepted suggested edits increased.

Pitfall: meticulous user feedback is plentiful. Our original hope was to study thumbs up/thumbs down user feedback as an additional model signal producing improved predictions. Unfortunately, developers do not often provide positive feedback explicitly: we noticed that most accepted suggested edits were not accompanied by a thumbs-up click, and there were few thumbs-up clicks for not-applied edits. Thankfully, in this case, applied edits constitute strong, indirect positive feedback. For negative feedback, we found a mix of "comment is not useful" and "suggested edit is not useful" clicks, which made the data very noisy, since the design of our feedback mechanism left room for ambiguous responses. In the end, we found the actual numbers of thumbs up/thumbs down relatively uniformative. However, a developer dissatisfied with the assistant's prediction has the option to open a bug report for the assistant, which collects automatically the changelist context, the reviewer comment, and the incorrect ML-suggested edit, as well as an optional explanation why this suggested edit is wrong. We received numerous such bug reports, which did help us and contributed to our curated evaluation dataset.

6 SYSTEM IMPACT

We now turn to evaluations, both qualitative and quantitative, of our code-review comment-resolution assistant in practice.

6.1 Quantitative Results

In the previous sections, we described how we iterated on model quality and usability, and reported relevant metric improvements for these iterations.

However, ultimately a primary goal for any assistance tool is to increase productivity. One metric we use to gauge the positive impact of our assistant on productivity is acceptance rate, the fraction of all code-review comments that are resolved by the assistant; this measures, out of all (non-automated) comments left by human reviewers, what fraction received an ML-suggested edit that the author accepted and applied directly to their changelist.

Table 1 shows results from the assistant's deployment, starting with the first milestone version (V1), which lacked reviewer preview and required a click to view suggested edits. The bottom V1 row shows that out of all code-review comments, 4.9% were resolved by an ML-suggested edit (with a 75–25 split between Critique and the IDE). Weekly aggregate acceptance rates over the 3-month deployment to 50% of Google engineers range between 3.9% and 5.5%. As has been previously published, tens of millions of code-review comments are left by Google developers every year [19], which means that an almost 5% ML-assisted comment resolution is a considerable contribution to Google's total engineering productivity.

The table also breaks down the fraction of comments that go through all the way to ML-assisted resolution. Around 34% receive predictions given the tuned model—note that this is a model tuned for 50% minimum precision. Of those predictions, A little fewer than a third are previewed by the author; for the rest, the author chose to push back (e.g., question the reviewer's comment), or perhaps thought the comment would be easy enough to address without even previewing the suggestion. Most of the previews were done in Critique and far fewer in the IDE (roughly a 5-to-1 ratio), which can be explained by Critique being the first-pass UI for receiving reviewer comments, only switching to the IDE when the review is

Version	Stage	(%) of total	(%) of previous step
	Incoming comments	100.0	100.0
V1	Confident predictions	34.3	34.3
	Previewed by author	10.7	31.3
	Applied by author	4.9	45.6
V2	Confident predictions	49.0	49.0
	Accepted by reviewer	33.1	63.6
	Previewed by author[a]	10.7	34.5
	Applied by author	7.5	69.5

[a]The concept of author preview is less significant in V2. The author automatically sees a small preview and can "click-to-view" full suggested edits. This full view either shows the "Apply" button or informs about an edit that requires a three-way merge. Almost all not-applied previews in V2 denote an edit that required a three-way merge to be applied.

Table 1: Comment attrition on the way to ML-assisted resolution. "V1" is the first milestone version (click-to-view suggested edits, no reviewer preview). "V2" is the second milestone version (reviewer preview, always show suggested edits to authors).

received while other coding is in progress, or the edit requires a 3-way merge to be applied. Finally, out of all previewed suggested edits, over 45% are accepted by the author and applied to code.

The more recent V2 milestone version (reviewer preview, no click to view edit) showed promising results during its beta deployment to a smaller fraction of Google engineers over 8 weeks, and was recently deployed to the full population replacing the V1 version globally. Table 1, under the V2 rows, shows that out of all code-review comments in this population, 7.5% were resolved by an ML-suggested edit (with a 76–24 split between Critique and the IDE). Weekly aggregate acceptance rates since full deployment started ranged between 5.4% and 7.5% with a standard deviation of about 0.9 percentage points. This acceptance rate is considerably higher than that of the V1 assistant; acceptance rate reached as much as 9.4% during the V2 beta, but stabilized around 7.5% after deployment covered the full developer population. Different programming languages see different acceptance rates. Among the popular languages, acceptance rate is 9.5% for Java, 7.5% for C++, and 7.1% for Python, and these numbers vary similarly to the above aggregate from week to week. We have no conclusive explanation why Python acceptance rate is lower than, say, Java, except that different popular languages are used for different purposes (e.g., building back-end services versus ML training scripts) and receive comments of different types.

Also looking at the breakdown (for the V2 table rows), around half of all eligible comments receive predictions—note that this model is tuned for 40% minimum precision, since a reviewer will have a chance to reject the suggestion. Of those predictions, over 63% are accepted by the reviewer and attached to the comment to be sent to the author. 34% of those suggested edits are previewed by the author, and of those previewed, 70% are accepted and applied to code. The higher "yield" of suggested edits as we proceed through the stages of the V2 assistant demonstrates the benefits of more reviewer control; we observe that reviewers do some "prompt engineering" on their comment text, to get the desired suggested edit. The higher yield also demonstrates the benefits of higher discoverability, and the assurance that a human reviewer has rejected inapplicable ML-suggested edits before the author receives them.

6.2 Qualitative Results

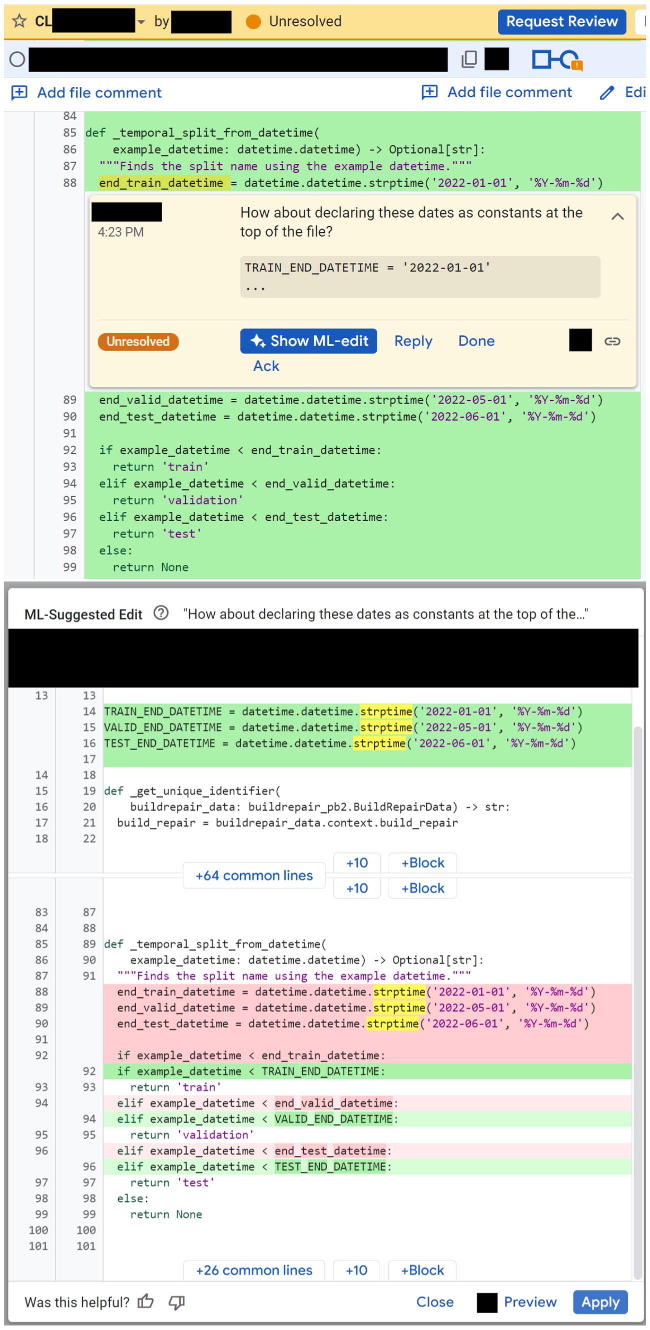
In this section, we look at some selected examples of detailed rewrites produced by the assistant. We observe ML-suggested edits addressing a wide range of reviewer comments in production, including simple localized refactoring, changes that are spread across dozens of lines of code, use of standard libraries accompanied by imports, unit-test generation, etc.

Figure 11 illustrates a refactoring edit that hoists some string literals strewn throughout the code into separate constants. The rewrite also moves some formatting code into the constant definitions, which addresses the reviewer's suggestion, but solves the problem in a more comprehensive way than what was proposed. It is interesting that the assistant is collaborating with the reviewer here: the reviewer suggests one way to address the issue and the assistant expands it into an even better rewrite.

The second example in Figure 12 addresses a fairly open-ended comment by the reviewer with a comprehensive code rewrite, introducing the idiomatic Python `enum` primitive, and importing the relevant standard library in the process. Although the reviewer was ambivalent about the right way to address their concern, the suggested edit demonstrates the detailed alternative for the author, leaving it up to them to make a choice (in this case, although the suggestion addressed the reviewer's suggestion, the author chose not to apply the edit).

In the final example in Figure 13, the model generates a new unit test to address the reviewer's suggestion. It mimics the pre-existing test pattern (from the location of the comment), while changing the expected semantics as described in the reviewer's comment. Additionally, the model suggests an apt test name, reflecting intended test semantics. Although different assistants might be required to handle more ambiguous and comprehensive test improvements, we have found that simple unit tests can be usefully suggested by the comment-resolution assistant.

Early feedback about the assistant in internal message boards is enthusiastic, including characterizations such as "sorcery!", "magic!", "impressive". Although the new version V2, in which suggested edits are presented as the reviewer is typing a comment, has only been deployed to 100% of the population for a relatively limited time, we have received delighted reports demonstrating that *just the location and the initial sentiment* of the reviewer's comment can lead to helpful suggested edits, for both parties involved. In one example, the reviewer had barely typed "I believe the recommended way [cursor]" with respect to four lines of code, and the model already suggested the refactoring that the reviewer intended, with no further prompting. In a similar cluster of examples, reviewers who remember there is a library for what they are about to suggest but do not recall the exact library or API will start writing the comment, e.g., "there is a library for [cursor]" and the suggested edit actually reminds them of the relevant library and the associated rewrite. In both cases the reviewer preview helps reviewers save



```
def _temporal_split_from_datetime(
    example_datetime: datetime.datetime) -> Optional[str]:
  """Finds the split name using the example datetime."""
  end_train_datetime = datetime.datetime.strptime('2022-01-01', '%Y-%m-%d')
```

[redacted] 4:23 PM

How about declaring these dates as constants at the top of the file?

```
TRAIN_END_DATETIME = '2022-01-01'
...
```

```
  end_valid_datetime = datetime.datetime.strptime('2022-05-01', '%Y-%m-%d')
  end_test_datetime = datetime.datetime.strptime('2022-06-01', '%Y-%m-%d')

  if example_datetime < end_train_datetime:
    return 'train'
  elif example_datetime < end_valid_datetime:
    return 'validation'
  elif example_datetime < end_test_datetime:
    return 'test'
  else:
    return None
```

"How about declaring these dates as constants at the top of the..."

[redacted]

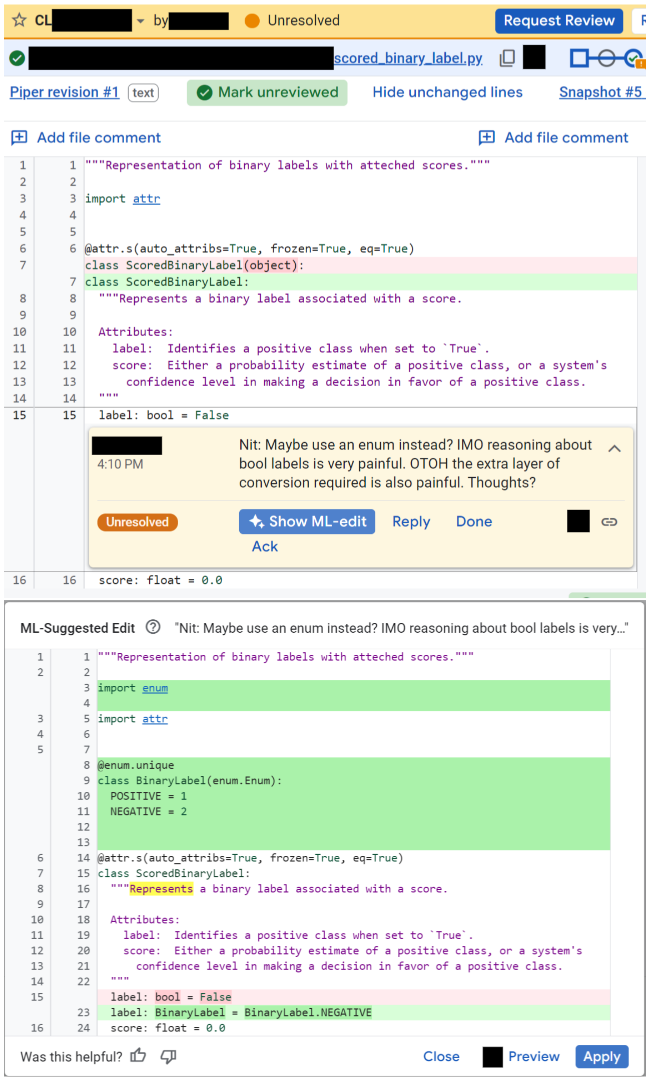
```

TRAIN_END_DATETIME = datetime.datetime.strptime('2022-01-01', '%Y-%m-%d')
VALID_END_DATETIME = datetime.datetime.strptime('2022-05-01', '%Y-%m-%d')
TEST_END_DATETIME = datetime.datetime.strptime('2022-06-01', '%Y-%m-%d')


def _get_unique_identifier(
    buildrepair_data: buildrepair_pb2.BuildRepairData) -> str:
  build_repair = buildrepair_data.context.build_repair

```

```


def _temporal_split_from_datetime(
    example_datetime: datetime.datetime) -> Optional[str]:
  """Finds the split name using the example datetime."""
  end_train_datetime = datetime.datetime.strptime('2022-01-01', '%Y-%m-%d')
  end_valid_datetime = datetime.datetime.strptime('2022-05-01', '%Y-%m-%d')
  end_test_datetime = datetime.datetime.strptime('2022-06-01', '%Y-%m-%d')

  if example_datetime < end_train_datetime:
  if example_datetime < TRAIN_END_DATETIME:
    return 'train'
  elif example_datetime < end_valid_datetime:
  elif example_datetime < VALID_END_DATETIME:
    return 'validation'
  elif example_datetime < end_test_datetime:
  elif example_datetime < TEST_END_DATETIME:
    return 'test'
  else:
    return None


```

Figure 11: Example: replace values with constants.

time reviewing, since it makes their comment concrete and avoids lookups, and it also saves the author's time, since the suggestion is actionable and pre-vetted by the reviewer.

The extensive use of the tool in the past year within the company has demonstrated non-trivial, detailed, comprehensive suggested edits that have surprised and delighted developers.



```
"""Representation of binary labels with atteched scores."""

import attr


@attr.s(auto_attribs=True, frozen=True, eq=True)
class ScoredBinaryLabel(object):
class ScoredBinaryLabel:
  """Represents a binary label associated with a score.

  Attributes:
    label:  Identifies a positive class when set to `True`.
    score:  Either a probability estimate of a positive class, or a system's
      confidence level in making a decision in favor of a positive class.
  """
  label: bool = False
```

[redacted] 4:10 PM

Nit: Maybe use an enum instead? IMO reasoning about bool labels is very painful. OTOH the extra layer of conversion required is also painful. Thoughts?

```
  score: float = 0.0
```

"Nit: Maybe use an enum instead? IMO reasoning about bool labels is very..."

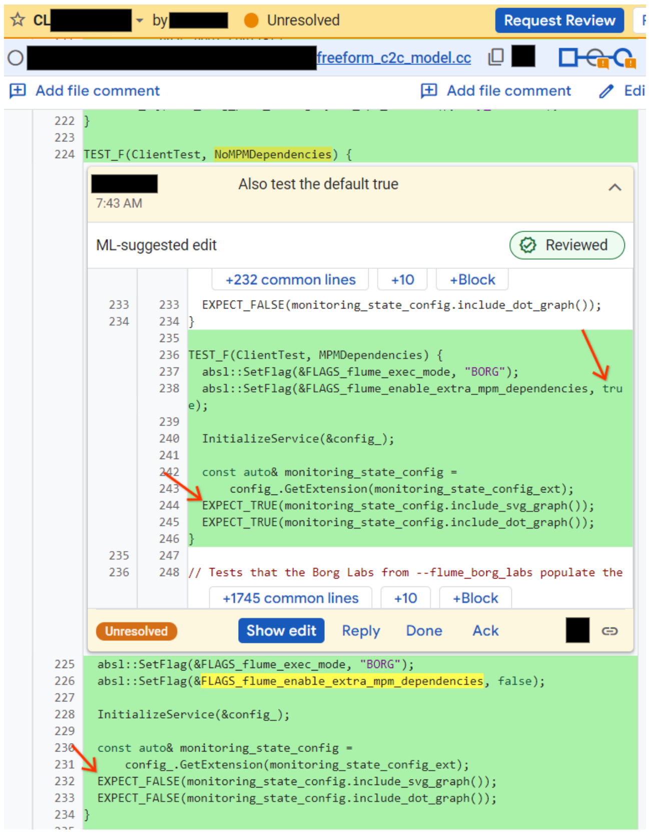
```
"""Representation of binary labels with atteched scores."""

import enum

import attr


@enum.unique
class BinaryLabel(enum.Enum):
  POSITIVE = 1
  NEGATIVE = 2


@attr.s(auto_attribs=True, frozen=True, eq=True)
class ScoredBinaryLabel:
  """Represents a binary label associated with a score.

  Attributes:
    label:  Identifies a positive class when set to `True`.
    score:  Either a probability estimate of a positive class, or a system's
      confidence level in making a decision in favor of a positive class.
  """
  label: bool = False
  label: BinaryLabel = BinaryLabel.NEGATIVE
  score: float = 0.0
```

Figure 12: Example: replace a Boolean label with an enumeration, also including relevant imports. Note the conversational language used by the reviewer.

7 RELATED WORK

There is quite a bit of recent work on predicting code review comments from code / code changes, but relatively less work on generating code edits to address comments, or experiences about such deployed systems.

CodeReviewer [10] is perhaps the closest recent result to our assistant. It seeks to automate the identification of lines that merit a comment, the generation of that comment, and a suggested refactoring to address the issue. It is trained on 8M GitHub pull requests to further pre-train and fine-tune a code Large Language Model. The reported results focus on the model itself and ablative studies, showing promising benchmark evaluations. Our approach is similar in principle, with some differences. First, our focus in this paper is on *deploying* an assistant based on a comment-resolution



```
}

TEST_F(ClientTest, NoMPMDependencies) {
```

[redacted]
7:43 AM
Also test the default true

```
  EXPECT_FALSE(monitoring_state_config.include_dot_graph());
}

TEST_F(ClientTest, MPMDependencies) {
  absl::SetFlag(&FLAGS_flume_exec_mode, "BORG");
  absl::SetFlag(&FLAGS_flume_enable_extra_mpm_dependencies, tru
e);

  InitializeService(&config_);

  const auto& monitoring_state_config =
      config_.GetExtension(monitoring_state_config_ext);
  EXPECT_TRUE(monitoring_state_config.include_svg_graph());
  EXPECT_TRUE(monitoring_state_config.include_dot_graph());
}

// Tests that the Borg Labs from --flume_borg_labs populate the
```

```
  absl::SetFlag(&FLAGS_flume_exec_mode, "BORG");
  absl::SetFlag(&FLAGS_flume_enable_extra_mpm_dependencies, false);

  InitializeService(&config_);

  const auto& monitoring_state_config =
      config_.GetExtension(monitoring_state_config_ext);
  EXPECT_FALSE(monitoring_state_config.include_svg_graph());
  EXPECT_FALSE(monitoring_state_config.include_dot_graph());
}
```

Figure 13: Example: generate a new unit test that flips a command-line flag from false to true, and adapts the test expectations.

ML model, and realizing its productivity impact. Second, we train on a more homogeneous distribution of code-review examples: code review is a fundamental aspect of the Google engineering culture, yielding a high volume of high-quality training data about a more uniform codebase. Third, DIDACT [12], our underlying ML framework, trains on a broader set of data sources and tasks (code, fine-grained code edits, code review, build repair, refactoring operations). PLUR [3] was our earlier research effort towards training multi-task models of diverse code activities. Finally, since this work is evaluated on end-user acceptance, rather than strictly on off-line accuracy results, accuracy metrics are not directly comparable; informationally, our recall@50 is over 66%, which is a competitive metric to the CodeReviewer metrics on its evaluation datasets.

Much ML research has focused on predicting code edits from some natural-language or semi-structured prompt, with applications to code review. Some notable examples include CoditT5 [27] and Graph2Edit [26]. In contrast, generative-ML research has sought to summarize problems in code, producing comments that could be used for code review. For example, Tufano et al. [22] and the more recent AUGER [8] both produce natural-language text about a code context and some review range and tags drawn from that code context. Such prior work shows that automatic review comments tend to be less helpful than user review comments—which, in part, motivates our focus on comment resolution rather than comment generation—but that direction of assistive code review shows some promise (e.g., AUGER found that 29% of automatic review comments are still considered useful, in their human study). LLaMa-Reviewer [11] is a very recent addition in this space, using the LLaMa large language model [21] to formulate a number of code-review related tasks, with a focus on parameter-efficient tuning, and showing impressive performance on the CodeReviewer and AUGER datasets, especially with respect to comment generation.

Resolving *tool*-generated comments or annotations using ML is an active sub-topic of program-repair research (e.g., DeepFix [6]), including our own prior work on *build repair* [13, 20] and Getafix [1]. Most of this work considers only stylized error messages generated by linters, static analyzers, or compilers. The kinds of code transformations expected in such cases are considerably narrower. By contrast, this paper aims to produce code transformations where the intent is expressed in natural language.

8 CONCLUSION AND NEXT STEPS

In this article, we introduced an ML-assistance feature to reduce the time spent on resolving code-review comments at Google. At the moment, the assistant is deployed to all Google engineers and 7.5% of all code-review comments are addressed with applied ML-suggested edits.

We are working on improvements throughout the whole stack, but also creating APIs to integrate this in-situ modification of code to address review comments in other tools and modalities.

Beyond code review, we are continuing along the same line of work to bring ML assistance to various stages of the software engineer's journey, from design and implementation all the way to deployment, maintenance, bug finding, and repair.

ACKNOWLEDGMENTS

This is the work of many people in Google Core Systems & Experiences team, Google Research, and DeepMind. We would like to thank all of our team members for their key contributions and useful advice, including Alberto Elizondo, Ambar Murillo, Amit Patel, Ballie Sandhu, Boris Bokowski, Brett Wiltshire, Chenjie Gu, Chris Gorgolewski, David Choi, David Tattersall, Dima Sutygin, Emily Johnston, Fredde Ribeiro, Henryk Michalewski, Hermann Loose, Jessica Ko, Jiquan Ngiam, Jonas Mattes, Juanjo Carin, Kathy Nix, Katja Gruenwedel, Kensen Shi, Kristóf Molnár, Kyle Lippincott, Laurent Le Brun, Luis C. Cobo, Luka Kalinovcic, Madhura Dudhgaonkar, Marc Brockschmidt, Marc Rasi, Marcus Revaj, Mariana Stariolo, Matt Frazier, Franjo Ivančić, Małgorzata Salawa, Manushree Vijayvergiya, Marko Ivanković, Markus Kusano, Mehdi Ghissassi, Michael Sloan, Mingpan Guo, Niranjan Tulpule, Ole Rehmsen, Paige Bailey, Peter Josling, Raphael von der Gruen, Rodrigo Damazio Bovendorp, Sara Qu, Sara Toth, Sara Wiltberger, Sascha Varkevisser, Savinee Dancs, Stephanie Tang, Stoyan Nikolov, Tobias Welp, Vahid Meimand, Vaibhav Tulsyan, Yujia Li, and Zoubin Ghahramani.

REFERENCES

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[3] Zimin Chen, Vincent J Hellendoorn, Petros Maniatis, Pascal Lamblin, Pierre-Antoine Manzagol, Danny Tarlow, and Subhodeep Moitra. PLUR: A Unifying, Graph-Based View of Program Learning, Understanding, and Repair. In *Thirty-fifth Conference on Neural Information Processing Systems (NeurIPS 2021)*, 2021.

[4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling language modeling with Pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[5] Alexander Frömmgen and Lera Kharatyan. Resolving code review comments with ML. *Google Research AI Blog*, May 2023. https://blog.research.google/2023/05/resolving-code-review-comments-with-ml.html.

[6] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[7] Ciera Jaspan, Matthew Jorde, Carolyn Denomme Egelman, Collin Green, Ben Holtz, Edward K. Smith, Maggie Morrow Hodges, Andrea Marie Knight Dolan, Elizabeth Kammer, Jillian Dicker, Caitlin Harrison Sadowski, James Lin, Lan Cheng, Mark Canning, and Emerson Murphy-Hill. Enabling the study of software development behavior with cross-tool logs. *IEEE Software*, Special Issue on Behavioral Science of Software Engineering, 2020.

[8] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. AUGER: automatically generating review comments with pre-training models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1009–1021, 2022.

[9] Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. StarCoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. Reproducibility Certification.

[10] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1035–1047, New York, NY, USA, 2022. Association for Computing Machinery.

[11] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. LLaMA-Reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *34th IEEE International Symposium on Software Reliability Engineering (ISSRE 2023)*, 2023.

[12] Petros Maniatis and Daniel Tarlow. Large sequence models for software development activities. *Google Research AI Blog*, May 2023. https://blog.research.google/2023/05/large-sequence-models-for-software.html.

[13] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. DeepDelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 925–936, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Rachel Potvin and Josh Levenberg. Why Google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[15] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[16] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 202–212, New York, NY, USA, 2013. Association for Computing Machinery.

[17] Adam Roberts, Hyung Won Chung, Gaurav Mishra, Anselm Levskaya, James Bradbury, Daniel Andor, Sharan Narang, Brian Lester, Colin Gaffney, Afroz Mohiuddin, Curtis Hawthorne, Aitor Lewkowycz, Alex Salcianu, Marc van Zee, Jacob Austin, Sebastian Goodman, Livio Baldini Soares, Haitang Hu, Sasha Tsvyashchenko, Aakanksha Chowdhery, Jasmijn Bastings, Jannis Bulian, Xavier Garcia, Jianmo Ni, Andrew Chen, Kathleen Kenealy, Kehang Han, Michelle Casbon, Jonathan H. Clark, Stephan Lee, Dan Garrette, James Lee-Thorp, Colin Raffel, Noam Shazeer, Marvin Ritter, Maarten Bosma, Alexandre Passos, Jeremy Maitin-Shepard, Noah Fiedel, Mark Omernick, Brennan Saeta, Ryan Sepassi, Alexander Spiridonov, Joshua Newlan, and Andrea Gesmundo. Scaling up models and data with `t5x` and `seqio`. *Journal of Machine Learning Research*, 24(377):1–8, 2023.

[18] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open foundation models for code, 2023.

[19] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. Modern code review: A case study at Google. In *International Conference on Software Engineering, Software Engineering in Practice track (ICSE SEIP)*, 2018.

[20] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. Learning to fix build errors with Graph2Diff neural networks. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 19–20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. LLaMa: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[22] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanykz, and Gabriele Bavota. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 163–174. IEEE, 2021.

[23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[24] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[25] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google*. O'Reilly Media, Inc., 2020. Available at https://abseil.io/resources/swe-book.

[26] Ziyu Yao, Frank F. Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. Learning structural edits via incremental tree transformations. In *International Conference on Learning Representations*, 2021.

[27] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. CoditT5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.