AI-Assisted Assessment of Coding Practices in Modern Code Review

Manushree Vijayvergiya
manushree@google.com
Google
Zurich, Switzerland

Małgorzata Salawa
magorzata@google.com
Google
Zurich, Switzerland

Ivan Budiselić
ibudiselic@google.com
Google
Zurich, Switzerland

Dan Zheng
danielzheng@google.com
Google
Mountain View, USA

Pascal Lamblin
lamblinp@google.com
Google
Montreal, Canada

Marko Ivanković
markoi@google.com
Google
Zurich, Switzerland

Juanjo Carin
juanjocarin@google.com
Google
Sunnyvale, USA

Mateusz Lewko
mlewko@google.com
Google
Zurich, Switzerland

Jovan Andonov
jandonov@google.com
Google
Zurich, Switzerland

Goran Petrović
goranpetrovic@google.com
Google
Zurich, Switzerland

Daniel Tarlow
dtarlow@google.com
Google
Montreal, Canada

Petros Maniatis
maniatis@google.com
Google
Mountain View, USA

René Just*
rjust@cs.washington.edu
University of Washington
Seattle, USA

ABSTRACT

Modern code review is a process in which an incremental code contribution made by a code author is reviewed by one or more peers before it is committed to the version control system. An important element of modern code review is verifying that code contributions adhere to best practices. While some of these best practices can be automatically verified, verifying others is commonly left to human reviewers. This paper reports on the development, deployment, and evaluation of AutoCommenter, a system backed by a large language model that automatically learns and enforces coding best practices. We implemented AutoCommenter for four programming languages (C++, Java, Python, and Go) and evaluated its performance and adoption in a large industrial setting. Our evaluation shows that an end-to-end system for learning and enforcing coding best practices is feasible and has a positive impact on the developer workflow. Additionally, this paper reports on the challenges associated with deploying such a system to tens of thousands of developers and the corresponding lessons learned.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

*Work done at Google.

AIware '24, July 15–16, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0685-1/24/07
https://doi.org/10.1145/3664646.3665664

KEYWORDS

Artificial Intelligence, Code Review, Coding Best Practices

ACM Reference Format:
Manushree Vijayvergiya, Małgorzata Salawa, Ivan Budiselić, Dan Zheng, Pascal Lamblin, Marko Ivanković, Juanjo Carin, Mateusz Lewko, Jovan Andonov, Goran Petrović, Daniel Tarlow, Petros Maniatis, and René Just. 2024. AI-Assisted Assessment of Coding Practices in Modern Code Review. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware '24), July 15–16, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3664646.3665664

1 INTRODUCTION

Modern code review [21, 23] (compared to holistic code review [8]) has grown organically over the years in open-source and industrial settings. A set of common peer-review criteria have emerged [5, 20, 21], which include coding best practices. Many companies, projects, and even programming languages formally define them in the form of "style guides" [1–4] that commonly cover the following aspects:

- *Formatting*: line limits, use of whitespaces and indentation, placement of parentheses and brackets, etc.;
- *Naming*: capitalization, brevity, descriptiveness, etc.;
- *Documentation*: expected placement and content of file-level, function-level, and other comments;
- *Language features*: use of specific language features in different (code) contexts;
- *Code idioms*: use of code idioms to improve code clarity, modularity, and maintainability.

Developers generally report high satisfaction with modern code review processes [23, 28]. One of their main benefits is the learning experience for code authors who are not familiar with the codebase, specific language features, or common code idioms. During a review, an expert developer educates the code author on best practices, in

addition to reviewing (and learning about) the code contributions and their implications.

Static analysis tools such as linters [15] can automatically verify that code adheres to some best practices (e.g., formatting rules), and some tools can even automatically fix violations. However, nuanced guidelines or those with exceptions are difficult to automatically verify in their entirety (e.g., naming conventions and justified deviations in legacy code), and some guidelines cannot be captured by precise rules at all (e.g., clarity and specificity of code comments) and rely on human judgement and collective developer knowledge. As a result, it is generally expected that human reviewers check code changes for best practice violations.

The biggest cost of the code-review process is the time required, especially from expert developers. Even with significant automation in place, and keeping the process as lightweight as possible, a developer can easily dedicate several hours daily to this task [23].

Recent advances in machine learning, capabilities of large language models (LLMs) in particular, suggest that LLMs are suitable for code-review automation (e.g., [11, 16, 17, 24–26]). However, the software engineering challenges around deploying an end-to-end system at scale remain unexplored. Likewise, extrinsic evaluations of such systems on overall efficacy and user acceptance are missing.

This paper investigates whether it is possible to partially automate the code-review process, specifically the detection of best practice violations, thereby providing timely feedback for code authors and allowing reviewers to focus on overall functionality. Specifically, this paper reports on our experience of developing, deploying, and evaluating AutoCommenter—an automated code-review assistant—in an industrial setting at Google, where it is currently used by tens of thousands of developers every day.

In summary, the contributions of this paper are:

- A general architecture of an LLM-based code-review assistant system (section 3).
- A description of tool calibration and deployment to tens of thousands of developers (section 4).
- An evaluation of the system (section 5).
- A summary and discussion of lessons learned (section 6).

2 BACKGROUND

AutoCommenter was developed in a large industrial setting at Google. The modern code review practices at Google are similar to those of other industrial and open source projects [23].

2.1 Code Review Process

The code review process at Google is well established, change-based, and tool-assisted. Ivanković et al. [12] and Petrović et al. [18] provide a detailed summary of the process. Each change to the codebase must be reviewed by at least one other developer. Every day, tens of thousands of changes to the codebase go through the review process and tens of thousands of developers participate in the process, as both code authors and reviewers.

Authors and reviewers exchange comments through the code review system, and a review progresses through snapshots of files affected by the change. Each reviewer comment is attached to a specific line and column range in a specific file snapshot. To resolve a comment, the author typically modifies the file in their local copy and exports a new snapshot for the next round of code review. When the author and all reviewers are satisfied and no automated analysis is blocking the merge, the code is merged into the codebase.

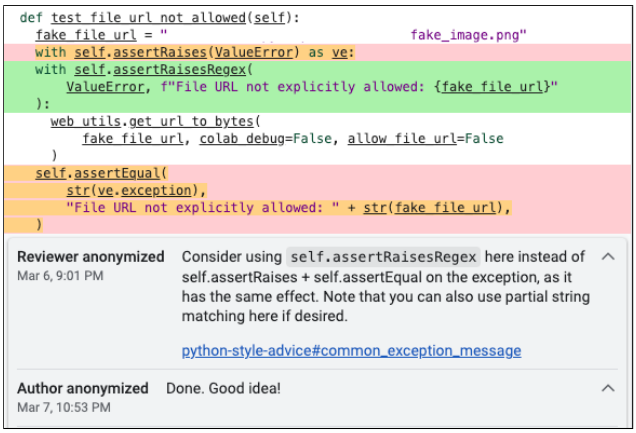

Figure 1: Example comment posted by a human reviewer.

The most expensive part of the code review process is the time spent by code authors and reviewers "shepherding" a change (from initial coding, through addressing reviewer comments and ensuring all automated analyses pass, to finally merging the change into the codebase). While the process is optimized with automated systems analyzing the code before the review (notably automatic code formatting without human intervention), code reviews still cost thousands of developer-years per year. Thus, even single-digit percentage savings translate into significant business impact.

2.2 Best Practices

A *best practice* is a specific use of programming language that is considered superior, and a *best practice document* describes how it should be applied and what benefits it brings. *Best practice URL* refers to a best practice document or specific section therein, and *best practice violation* refers to a specific piece of code that does not adhere to a best practice, but can be changed to do so. If clear from context, we use the terms *URL* and *violation* to refer to best practice URL and best practice violation, respectively.

Google's central code repository contains code in many languages, with C++, Java, Python and Go exceeding 100 million lines each [19]. For 15 different languages there are formal style guides readily available to all developers. Many of these languages have additional language primers, documentation for core libraries, and tip-of-the-week style newsletters. While these materials are not as strictly enforced as style guides, they are frequently referenced in code reviews. Some languages boast hundreds of pages of such documentation. Both code authors and reviewers are expected to verify that the code follows all best practices.

A formal mechanism called "readability", introduced more than a decade ago, ensures that best practices are followed consistently. Dedicated style experts in a given language, called "readability mentors", guide inexperienced developers towards proficiency in the language [23]. Readability mentors commonly summarize a best practice in a few sentences and at the end of the comment include a URL for the change author as a reference. Figure 1 shows an example of a comment posted by a readability mentor.

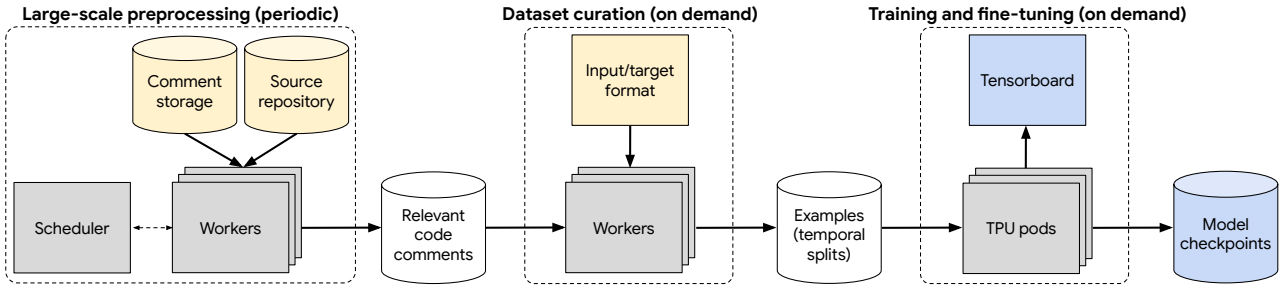

Figure 2: Architecture of the model-training pipeline.

The readability process has some drawbacks. For authors, it increases development time due to additional review rounds. For readability mentors, it can become a monotonous and time-consuming task. It requires mastering hundreds of evolving best practices, including the identification and deprecation of outdated rules, and documenting them (with relevant links) in the code-review system. Additionally, it requires tracking, sometimes through multiple iterations, ensuring that all violations have been remedied.

3 APPROACH

In response to the challenges described sections 2.1 and 2.2, we developed AutoCommenter, a code analysis tool that automatically detects best practice violations. It aims to provide timely feedback for code authors and to alleviate the need for manual best-practice reviews, thereby allowing reviewers to focus on code functionality.

3.1 Model and Task Definition

Automating best practice analysis requires a model that can represent source code, pinpoint violation locations, and identify the violated best practice. We target a text-to-text transformation using a traditional transformer approach based on T5, using T5X [22].

The best practice analysis is one task in a multi-task large sequence model. In addition to the standard pretraining task for T5, span denoising (predicting masked tokens), other tasks used to train this model include code-review comment resolution, next edit prediction, variable renaming, and build-error repair [9]. The training corpus consists of over 3 billion examples, of which the best practice analysis dataset contributes about 800k examples. The model was trained using the standard cross-entropy loss, typical for such models, and tuned to maximize the sequence accuracy metric, predicting the exact target text for each example.

For the best practice analysis, the **input** to the model is a task prompt and source code, and the **target** is a source code location and a URL for a best practice violation. The task prompt is formatted as a fixed-text code comment, using the programming language's appropriate commenting style. It describes the task in natural language and precedes the source code, which is a direct textual representation of one file. If the input exceeds the model context window, it is truncated. The location is a byte offset in the source code, and the URL references the violated best practice. A domain specific language defines the target format, and a special case is the "empty" target, if there are no violations. In addition to the target, the model outputs a confidence score ranging from 0 to 1.

Consider the following input/target example for the Go language.

Input

```
// [*] Task: Check language best practices.
// Package addition provides Add
package addition

// Return a sum
func Add(value1, value2 int) int {
        return value1 + value2
}
```

Target

```
INSERT 153 COMMENT https://go.dev/doc/comment#func
```

The first line of the input is the fixed-text task prompt; the rest is the source code. The target gives the location (byte offset 153 corresponds to the start of the `Add` function) and a `go.dev` URL, pointing to the exact part of the Go language style guide that the function comment violates (in this case, the usual practice of starting a comment with the function name). Note that the target may contain no, one, or multiple (concatenated) location-URL pairs, depending on the number of violations in the source code.

3.2 Model Training

Figure 2 shows the architecture of the model-training pipeline, which consists of three parts. We split dataset creation into two steps (preprocessing and curation) because the first step is significantly more expensive as it operates on a much larger amount of data. The output of the preprocessing step is agnostic to the model's input/target representation. This separation improves feature velocity by enabling quick iterations on example representations and other example-level adjustments. The preprocessing step uses a fault-tolerant scheduling system and periodically extracts relevant code comments to ensure that new data is readily available.

3.2.1 Large-scale preprocessing. The training examples are created from real code review data, but not all code comments are suitable for model training. Therefore, the preprocessing step, identifies *relevant code comments*—human authored comments that contain a URL pointing to a best practice document. For each comment, the preprocessing step then collects the corresponding source code and relevant metadata, including the comment's location in the source code and its creation time. The output of this step is a set of relevant code comments, each with all the data necessary for curating examples for model training.

3.2.2 Dataset curation. Dataset curation is a single, on-demand processing step, implemented as a Beam[1] pipeline. It converts each relevant code comment, based on the input/target format described in section 3.1, into the standard TensorFlow *Example* data structure.

3.2.3 Training and fine-tuning. The curated examples are used directly for model training and evaluation. We use the T5X framework [22] on a fleet of TPUs, store the model checkpoints every 1000 steps, and use Tensorboard for monitoring the training.

3.3 Model Selection

Two intrinsic evaluations on historical data inform our selection of a model checkpoint, confidence thresholds, and a decoding strategy. First, an evaluation on the validation and test datasets provides estimates of precision and recall on a per-file basis. Second, an evaluation on full historical code reviews provides an estimate of the total number of comments per code review, indicating how often developers would interact with AutoCommenter.

3.3.1 Evaluation on Validation and Test Datasets. We temporally split the dataset to ensure that the model has not been trained on future code-review snapshots of the code comments in the validation and test datasets. In our dataset, 85% of files have exactly one relevant code comment, 11% have two, and 4% have three or more. We define a prediction to be *correct* if the predicted code location(s) and URL(s) match the expected values, regardless of order.

Recall that the model provides a confidence score for each prediction, which introduces another parameter: a prediction can be suppressed if its confidence score is below some threshold t. We define $Precision_t$ as the number of correct predictions whose confidence score is greater than t divided by the number of all predictions whose confidence score is greater than t; we define $Recall_t$ analogously. These definitions allow us to estimate how many (in)correct results would be shown to a user, as a function of t. $Precision_t$ and $Recall_t$ are used for model checkpoint comparisons during training.

While this evaluation avoids data leakage and allow us to automatically evaluate model performance, it has a limitation: while it is reasonable to assume that the human comments for a given code-review snapshot are correct, they are not exhaustive. In other words, it is possible that the code in a given code-review snapshot could be improved according to multiple best practices, but a human reviewer did not post comments (with URLs) for all of them. This can happen for several reasons:

- *Missing references*: A reviewer may comment on an issue, but did not include a URL as a reference.
- *Selective commenting*: A reviewer may comment on an issue once, expecting the author to apply a fix throughout.
- *Varied expertise or focus*: A reviewer may not be familiar with all best practices, or simply choose not to comment on an issue in the context of a given code review (e.g., focusing only on changed code).

While most files in our dataset have only one relevant comment, anecdotal evidence based on manually inspecting "incorrect" predictions suggests that multiple best-practice comments are typically possible due to the reasons stated above. Given that our ground-truth data is incomplete, our precision and recall measures are noisy. Therefore, we employ a complementary evaluation, described next, to increase confidence in overall model performance.

3.3.2 Evaluation on Full Historical Code Reviews. To accurately gauge potential comment volume in a live setting, we evaluate AutoCommenter on a set of historical code reviews, using a specific model checkpoint and threshold. The predicted comments are not retroactively posted in the code review system, but rather logged in a database for analysis. This allows us to estimate the expected posting frequency—both at per-file and per-code-review granularity. Because developers interact with AutoCommenter for an entire set of code changes subject to code review, this evaluation is an important step before production deployment. As an added benefit, this step allows for further optimizations and assessment of posting frequencies for different user groups, programming languages, etc.

3.4 Inference Infrastructure

The core of AutoCommenter is a central best practice analysis service. This service takes as input one or more source files for analysis. For each file, it constructs a model input (section 3.1), encodes it in the standard TensorFlow *Example* data structure, and queries the model. The model itself is served by a model service that uses TensorFlow's *Example* data structure as a domain agnostic input-output format. Finally, the best practice analysis service performs a series of filtering steps (section 4), which suppress low-quality predictions, and returns the remaining predictions.

3.5 IDE and Code Review Integration

Developers interact with AutoCommenter's analysis service in two ways—directly through an IDE plugin, or indirectly through the code review system. The code review system is used by all developers at Google, and the IDE by almost all of them.

AutoCommenter's comments appear in the IDE as diagnostics marked with a blue curly underline, spanning the relevant code snippet. Hovering over the underlined code reveals the full comment with a concise summary of the best practice, including a clickable link to the relevant best practice document. This embedded information streamlines the workflow for developers by eliminating the need to switch between the IDE and a web browser for unfamiliar best practices. Since comments in the IDE need to be generated in real-time, we aim to generate comments with sub-second latency.

In the code review system, AutoCommenter runs after each update (i.e., on each new code-review snapshot), automatically posting comments if it detects any violations. Comments produced by automated tools are visually similar to comments produced by humans, but have a differently colored background.

Figure 3 shows an example comment generated and posted by AutoCommenter in the code review system. Note the thumbs up and thumbs down buttons (right), which authors and reviewers can click if they find a comment particularly useful or not. Also note the "Please fix" button (left), which is visible to reviewers. If clicked, a new comment is generated indicating that the reviewer believes the comment is significant and must be addressed before the code is merged into the codebase. These feedback buttons are standard in the code review system, present on all comments generated by automated tools (e.g., [7, 13]), and provide a signal for a tool's user acceptance. The IDE provides a similar feedback mechanism.

[1] https://beam.apache.org/

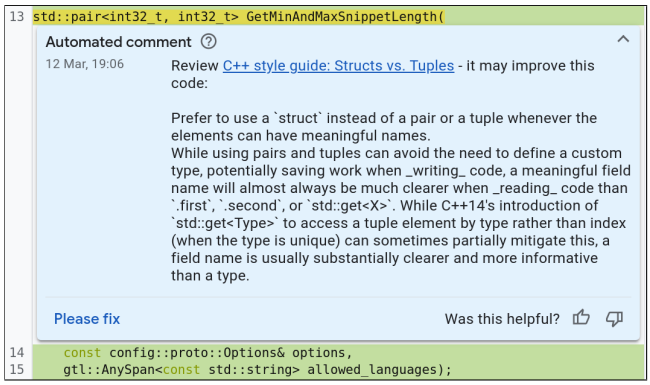

Figure 3: Example comment posted by AutoCommenter.

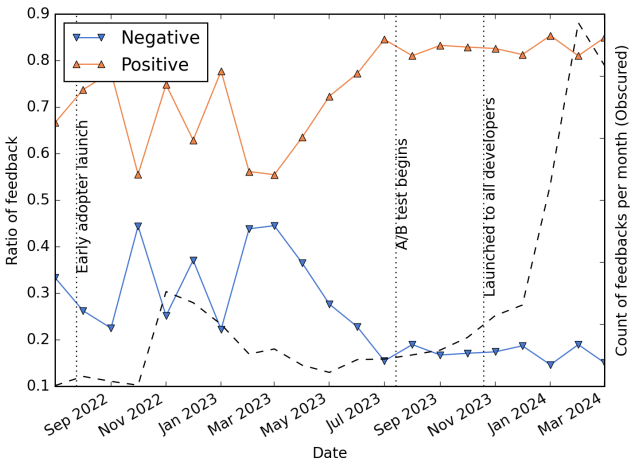

Figure 4: Developer feedback throughout deployment.

4 DEPLOYMENT

We deployed AutoCommenter to all developers at Google over a period of time between July 2022 and October 2023:

- **until Jul. 2022—teamfooding**: this paper's authors.
- **Jul. 2022—early adopters**: around 3 thousand volunteers.
- **Jul. 2023—A/B experiment**: about half of all developers.
- **since Oct. 2023—general availability**: all developers.

Note that due to industrial confidentiality reasons we are unable to disclose absolute numbers of code reviews, developers, files, comments, or distribution of duration of code reviews. We report on relative measures, where appropriate, and relevant trends.

We continuously evaluated and improved the performance of AutoCommenter, using an iterative refinement approach:

- Evaluation on historical data (section 3.3) to get directional insight into how well the model does at the task and to define thresholds and select a decoding strategy.
- Monitoring and analysis of user interaction and direct feedback through feedback buttons and issue reports.
- Targeted human evaluation based on patterns observed during other evaluation steps.

Figure 4 shows the ratio of positive to negative developer feedback on posted code review comments and IDE diagnostics over time. The dashed line shows the total count of feedback clicks developers provided per month. As is expected, this count is much lower during the early-adopter stage. Additionally, the volatility is higher in this stage because we actively refined AutoCommenter.

Recall the three feedback buttons within the code review system (figure 3), which allow developers to express positive and negative sentiment about a posted comment. We consider comments with a thumbs up or "Please fix" as positive, and comments with a thumbs down as negative; we define the *useful ratio* as the ratio of positive comments to all comments with feedback.

The remainder of this section describes specific observations and corresponding refinements that we made during deployment.

4.1 Selecting Threshold and Decoding Strategy

4.1.1 Threshold. During initial deployment we wanted to carefully manage the trust developers had in AutoCommenter and started with a high confidence threshold of $t = 0.98$. We manually sampled several hundred results and observed that around 80% of predictions below the threshold were still correct—that is, the false-negative rate was very high at $t = 0.98$. Additionally, we observed that predictions in Python showed a significantly different distribution of confidence scores, which were disproportionately impacted by thresholding. We conjecture that the training dataset composition (number of distinct URLs and URL frequencies) and specificity of the best practice documents are reasons, but leave a deeper investigation to future work. An attempt to deploy per-language thresholds proved ineffective as a single threshold per language still did not adequately capture the model's ability to correctly predict hundreds of diverse best practices. This led to a lack of diversity in predicted URLs as the model tended to produce higher scores for some URLs vs. others, irrespective of correctness. These observations led to the first major change to AutoCommenter: per-URL thresholds computed based on the intrinsic evaluation on the validation dataset.

4.1.2 Decoding. An evaluation using per-URL thresholds with greedy decoding on full historical code reviews revealed that AutoCommenter detects violations in 6% of all changed files. However, 80% of comments would have been posted on lines of code not modified by the author. Developers typically do not take action on unchanged code. Consequently, AutoCommenter filters generated comments on unchanged lines of code, reducing the ratio of comments in changed files to 1.3%. In order to increase this ratio, we experimented with different decoding strategies: greedy (default), beam search, top-k, and top-p sampling. We settled on beam search (generating $n = 4$ potential responses), which tripled the posting frequency to 3.9%. It also yielded a substantially higher URL diversity: the 10 most-frequently posted URLs accounted for 41% of all comments, compared to 80% for greedy search.

Latency is another important aspect when choosing a decoding strategy for deployment. While beam search increased the posting frequency and diversity, inference became noticeably slower (median latency of 2 seconds). Given that this latency is prohibitive for interactive use in the IDE, we ultimately decided to use beam search for the code review system and greedy search for the IDE.

4.2 Suppressing Outdated Best Practices

After launching AutoCommenter to around 3 thousand voluntary first-adopters, we noticed a large number of issues being filed by users within a few days. Many of these corresponded to a single URL[2], which describes best practices related to Python imports. However, the canonical source for some type names had changed in Python 3.9, and the best practice had also changed in early 2022. Since our training data stretches before 2022, it contained a number of best practice comments that were no longer applicable. We realized that this is a recurring pattern: as languages evolve, or new libraries are introduced, best practices evolve as well. One way of mitigating the problem is to filter out such data (whenever a rule changes), and retrain the model. This is however time and resource-consuming: it requires full data regeneration, model training, evaluation, and rollout. In the meantime, the "outdated" model needs to be either switched off, causing downtime of the system, or affected predictions need to be suppressed. Otherwise, the system could quickly lose developer trust. We opted for suppression of specific best-practice predictions, using conditional filtering (matching regular expressions on the source code) for two reasons. First, it can be dynamically deployed and immediately applied. Second, it allows for granular filtering of predictions.

4.3 Independent Rating of Selected Comments

After several months of early usage, we observed that the useful ratio plateaued at around 54%. To understand the reasons, identify areas for improvement, and prepare for a wider deployment, we conducted an independent human rating study in April 2023, analyzing a sample of around 370 posted comments that received developer feedback during our first-adopters deployment.

To gather diverse perspectives on the usefulness of comments we recruited 15 raters—developers from partner teams. We asked them to rate AutoCommenter's comments that received explicit user feedback. We did not show the original user feedback to the rater, to avoid biasing their evaluation. The raters assessed each comment's usefulness based on the linked best practice and the surrounding code. We instructed them to focus on comment correctness, but also whether the comment would be actionable to them as an author (e.g., would they resolve a comment that is technically correct but does not seem worth resolving in a specific instance). They were encouraged to provide free-form feedback on each comment.

The useful ratio from the rater evaluation was 60%, slightly higher than the 54% from the developer feedback on the same comments, but well below our target of 80% for wider deployment.

The most interesting finding from this study was that there were clear patterns of not useful comments. Here are some examples:

Several topics or complex topic: For example, one URL points to a section that describes multiple guidelines for interacting with the Python linter, including cases where it often triggers and ways to suppress it. An author may struggle to understand what specific guideline a posted comment is referring to and how to resolve it. Similarly, the guidance on writing good function documentation in C++ is a full page of dense text. Raters frequently noted a disconnect between a best practice (and AutoCommenter's concise summary) and the actual code, even when it contained a relevant violation.

[2]https://github.com/google/styleguide/blob/gh-pages/pyguide.md#22-imports

Importance of high-quality summaries: Raters often found that AutoCommenter's summary, which was generated by scraping the document source and sometimes missing, failed to adequately explain the relevance of the cited guideline to the comment/code.

Subjective and potentially contentious topic: One example is avoiding flags in library code. Flags can cause problems when used in libraries, but some libraries are specifically designed to have many features configurable via flags. Additionally, legacy code may not adhere to this guideline and reviewers will not enforce it. The model did not learn these nuances and sometimes predicted a violation when an author added a new flag to an existing library.

Systematic model error for some guidelines: One interesting example is a guideline that promotes the use of the member function *push_back* over *emplace_back* for C++ vectors when both functions can be used with the same arguments to achieve the same effect. The model had learned to predict this, but it would also predict it in cases where *emplace_back* is warranted, and also when an unrelated type had a member function called *push_back*.

Correct but low-value comments: A missing period at the end of a sentence in a code comment is often allowed by human reviewers. While technically correct, asking the author to go back to their IDE and fix the issue may provide net negative value.

The insights from the rater study informed two changes to AutoCommenter. First, the rater study identified 17 non-actionable URLs, whose suppression increased the historical useful ratio from 54% to 66% on developer feedback, and from 60% to 74% on rater feedback. We further analyzed comments linked to similar, unrated URLs and suppressed an additional 5. Second, we reviewed and manually updated summaries for all frequently posted URLs. Together, these changes were sufficient to reach our target useful ratio of 80% for the next stage of deployment.

4.4 A/B Experiment

In July 2023, we deployed AutoCommenter to about half of all developers in the context of an A/B experiment. We randomly assigned developers to a an experiment group (AutoCommenter enabled) and a control group (AutoCommenter disabled). We randomized based on the last few digits of the SHA256 hash of the developers email address, and we verified that both groups did not differ in size and composition, including distribution of tenure, seniority, programming languages and business units. We also confirmed that none of the variables measured during the experiment differed between the control and the experiment group before the experiment began. The comment posting frequency during the experiment was in line with expectations (section 4.1).

We did not detect any statistically significant change in any of the following: total duration of code reviews, time developers actively spent on the code review, the number of comment-response iterations between the author and the reviewer. We did, however, detect a slight improvement in coding speed. We conjecture that the reduction in context switches to documentation leads to this positive effect. We leave a deeper investigation for future work. Based on the results, we concluded that there are no adverse effects, and deployed AutoCommenter to all developers in October 2023.

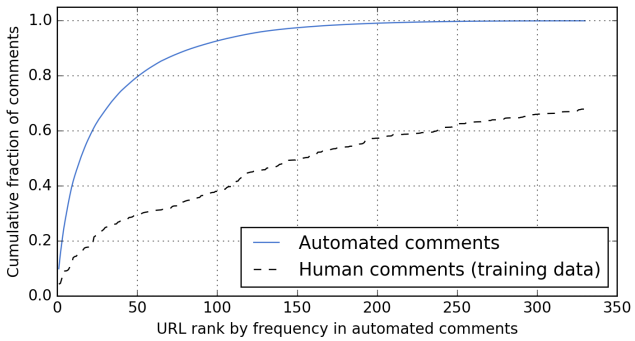

Figure 5: Cumulative distribution of comments per URL for the automated comments generated by AutoCommenter in production and human comments in the training data.

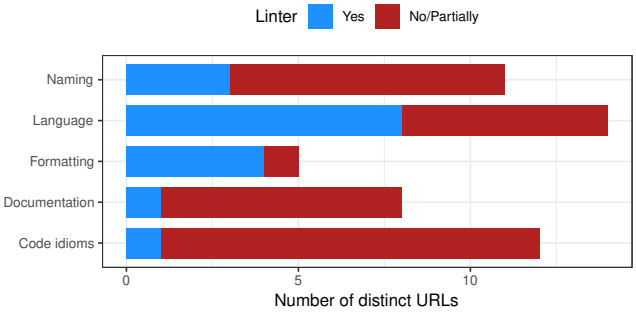

Figure 6: The top-50 most frequently predicted URLs categorized into types. Linter indicates whether a linter that detects a violation exists or can easily be built.

5 EVALUATION

Based on the useful ratio and user feedback gathered since March 2023, we conclude that developers are generally satisfied with the comments produced by AutoCommenter. We continuously refine our dataset preparation, thresholds, URL suppression and summarization by analyzing user feedback, to ensure that AutoCommenter delivers high positive impact on the developer workflow.

Beyond developer satisfaction, with several months in wide release to all of Google, we evaluated three additional aspects of AutoCommenter's performance:

(1) **Comment resolution:** How often do developers modify their code to resolve AutoCommenter's posted comments?
(2) **AutoCommenter vs. human comments:** How well do AutoCommenter's comments cover the best practice documents that human reviewers reference in their comments?
(3) **AutoCommenter vs. linters:** To what extent does AutoCommenter's output go beyond the capabilities of traditional static analysis tools?

5.1 Comment Resolution

Developers rarely give *explicit feedback* on AutoCommenter's comments by clicking the thumbs up/thumbs down buttons in the code review system and IDE, and the "Please fix" button in the code review system (figure 3): about 10% of automated comments in the code review system and 2% of diagnostics in the IDE received explicit feedback, which is comparable to other automated analyses at Google. At the same time, developers hover over approximately 50% of the AutoCommenter's IDE diagnostics, and prior work showed that developers often resolve automated comments without explicit feedback [18]. To assess how often developers resolve AutoCommenter's comments, we conducted an offline analysis, estimating the ratio of comments resolved by subsequent code changes.

To analyze comment resolution, we extracted historical changes focused on files with automated comments from AutoCommenter. For each, we extracted the initial snapshot where the comment was posted and the snapshot that the developer eventually merged into the codebase. Each comment spans a specific range of lines. We used an automated AST-based line mapping approach [18] between these snapshots, to identify comments that the model originally predicted on the first snapshot, but did not predict on the merged snapshot. Such pairs of snapshots indicated that a comment may have been resolved, but it is possible that unrelated code changes could have led to a specific comment no longer being predicted.

An automated analysis of 6000 snapshot pairs revealed that in 50% of cases the comment was absent from the submitted snapshot on the lines it was originally posted. We manually inspected a random sample of 40 such pairs. We found that in 80% of cases, a change made by the author directly resolved the issue described by the posted comment. Therefore, we estimate that the comment-resolution rate is about 40%, which is significantly larger than the ratio of comments with explicit positive feedback to all comments.

5.2 AutoCommenter vs. Human Comments

Figure 5 compares the cumulative distribution of comments (per unique URL to a best practice document) for the automated comments generated by AutoCommenter in production and human comments in the training data. The x-axis is the rank of the URL when all URLs ever used in automated comments are sorted by frequency. For example, the most frequently used URL has rank 1, and it accounts for 9.9% of all automated comments. This same URL appeared in 4.3% of the human created comments in the training data. In total, AutoCommenter has created comments for 330 distinct URLs. The set of URLs used by AutoCommenter covers 68% of historical human comments with a best practice URL. This is a good result: it demonstrates that AutoCommenter is not focusing on obscure best practices that are rarely referenced by reviewers.

On the other hand, despite utilizing beam search, URL diversity remains relatively low. The top-85 URLs make up 90% of comments created by AutoCommenter. The same set of URLs cover 35% of human comments with best practice URLs. Improving URL diversity and coverage of best practices in automated comments while maintaining accuracy and low latency is one of our top priorities.

5.3 AutoCommenter vs. Linters

To understand to what extent AutoCommenter provides value beyond linters that can efficiently and precisely check some of the best practices, we sampled the top-50 most frequently predicted violations—that is, the top-50 URLs in figure 5. For each sampled

URL, we inspected its best practice document and determined (1) the best-practice type (section 1) and (2) whether a linter that detects a corresponding violation exists or can be easily built. Specifically, three authors, each with over 10 years of experience in building static analysis tools, read the documentation and independently categorized the URLs. There were no disagreements on the best-practice type, but there were disagreements on whether a linter can be easily built for about 15% of URLs. The three authors resolved these disagreements through majority vote and discussion. Disagreements stemmed from ambiguous best practices, and those with multiple guidelines. For example, while checking the presence of code documentation is relatively straightforward, reasoning about justified exceptions and clarity of content may not.

Figure 6 shows the distribution of the 50 sampled URLs, broken down by type and whether violations can be detected by a linter. For 33/50 (66%) of these best practices, violation detection is beyond the scope of traditional static analysis.

6 LESSONS LEARNED

Based on our experience developing and deploying AutoCommenter, we summarize a few key lessons learned:

- **Complementing traditional analyses**: AutoCommenter's LLM-backed approach generates comments for 68% of best practices frequently referenced by human reviewers. Many of these are out of scope for traditional static analyses.
- **Intrinsic evaluation vs. real-world performance**: Intrinsic evaluations and real-world performance can diverge significantly: our intrinsic evaluation, using a dataset of real-world human comments together with a state of the art model architecture and training process, indicated a promising model, but our extrinsic evaluations and system improvements proved essential for a successful deployment.
- **Monitoring user acceptance is critical**: Even a few negative user experiences can erode trust in an automated system. Continuously monitoring and analyzing real-world feedback was crucial in detecting such instances and identifying remedies. In the case of AutoCommenter, a simple suppression mechanism was sufficient to strongly improve user acceptance to over 80% without major sacrifices in efficacy.

7 RELATED WORK

Johnson [15] introduced the C linter almost 50 years ago in 1977. In those 50 years, a considerable body of research on automated static analysis was produced: a recent literature review by Heckman and Williams [10] identified 17,571 papers. Many studies explore how developers interact with static analysis. Johnson et al. [14] explore challenges developers face when trying to use static analysis. The results of their study highlights the importance of good integration into existing developer workflows and the importance of developing and maintaining trust in the tool. Vassallo et al. [27] explore how developers interact with static analysis in different contexts, including coding and code review. They too find that integration into existing workflows plays a major role in developers willingness to use the tools and that high quality of results is extremely important. Beller et al. [6] studied usage of static code analysis in a large number of open source projects. Among other findings, they highlight that how automated analysis is and should be used varies based on the programming language.

In contrast, using machine learning for code analysis is a comparatively new and less understood field. A number of recent publications (e.g., Hong et al. [11], Li et al. [16], Li et al. [17], Thongtanunam et al. [24], Tufano et al. [25], and Tufano et al. [26]) report on model evaluations and propose tools for automated code review. While these models and the review comment generation task are very similar to the model presented in this paper, evaluations largely focused on historical datasets. As discussed in section 3.3.1 an intrinsic evaluation on only historical comments is somewhat limited and can sometimes fail to predict real-world performance. Another recent publication by Frömmgen et al. [9] presents an evaluation of a live system, but for the opposite task: creating code from comments rather than comments from code.

8 CONCLUSION

Verifying that code adheres to best practices is a common task in modern code review processes. While some best practices can be automatically verified with traditional tools such as linters, many require the knowledge and judgement of experienced developers, which requires time and effort.

This paper reports on our experience developing, deploying, and evaluating AutoCommenter, an LLM-backed code review assistant system. Specifically, it lays out the entire process from task and model design, over intrinsic evaluations and system calibrations, to a staged roll out and end-user evaluation.

The evaluation results show that it is feasible to develop an end-to-end system with capabilities well beyond traditional tools while achieving a high degree of end-user acceptance. These results are a promising first step towards the deployment of sophisticated code-review assistants and automated code reviews.

Our priority was to ensure a positive developer experience by designing AutoCommenter to have very high precision. While recall was not the primary focus, we recognize its significance and plan to explore what changes in the model and system architecture can improve recall. For example, the model we used in 2022 was state of the art at the time. However, it has a limited context window of 2048 tokens which suffices for only around 200 lines of code. Current state of the art models have context windows of tens of thousands of tokens during training and over a million tokens during inference. This leap opens up opportunities for new features and significant improvement in existing ones.

9 ACKNOWLEDGEMENTS

This work is the result of years of collaboration between teams in Google Core Systems and Google DeepMind. We are grateful for the support and advice of all our team members and leadership, including Alberto Elizondo, Alexander Frömmgen, Ballie Sandhu, Chandu Thekkath, Chris Gorgolewski, David Tattersall, Ilya Cherny, Jacob Austin, Katja Grünwedel, Kristóf Molnár, Lera Kharatyan, Luka Rimanić, Madhura Dudhgaonkar, Marc Brockschmidt, Marcus Revaj, Maxim Tabachnyk, Nina Chen, Niranjan Tulpule, Nitya Ramani, Paige Bailey, Pavel Sychev, Pierre-Antoine Manzagol, Quinn Madison, Roger Fleig, Satish Chandra, Savinee Dancs, Stoyan Nikolov, Subhodeep Moitra, and Vaibhav Tulsyan.

REFERENCES

[1] 2024. Google Style Guides. https://google.github.io/styleguide/. Accessed: 2024-03-15.

[2] 2024. Linux kernel coding style. https://www.kernel.org/doc/html/v4.10/process/coding-style.html. Accessed: 2024-03-15.

[3] 2024. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/. Accessed: 2024-03-15.

[4] 2024. Rust Style Guide. https://doc.rust-lang.org/nightly/style-guide/. Accessed: 2024-03-15.

[5] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. 712–721. https://doi.org/10.1109/ICSE.2013.6606617

[6] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[7] Zimin Chen, Małgorzata Salawa, Manushree Vijayvergiya, Goran Petrović, Marko Ivanković, and René Just. 2023. MuRS: Mutant Ranking and Suppression using Identifier Templates. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*. 1798–1808.

[8] M. E. Fagan. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15, 3 (1976), 182–211. https://doi.org/10.1147/sj.153.0182

[9] 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, and Petros Maniatis. 2024. Resolving Code Review Comments with Machine Learning. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[10] Sarah Heckman and Laurie Williams. 2011. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology* 53, 4 (2011), 363–387. https://doi.org/10.1016/j.infsof.2010.12.007 Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing.

[11] Yang Hong, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Aldeida Aleti. 2022. Commentfinder: a simpler, faster, more accurate code review comments recommendation. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 507–519.

[12] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code Coverage at Google. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 955–963.

[13] Marko Ivanković, Goran Petrović, Yana Kulizhskaya, Mateusz Lewko, Luka Kalinovčić, René Just, and Gordon Fraser. 2024. Productive Coverage: Improving the Actionability of Code Coverage. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[14] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[15] Stephen C Johnson. 1977. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

[16] Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo. 2022. Auger: Automatically generating review comments with pre-training models. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1009–1021.

[17] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. 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* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1035–1047. https://doi.org/10.1145/3540250.3549081

[18] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2023. Please fix this mutant: How do developers resolve mutants surfaced during code review?. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 150–161.

[19] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Communications of the ACM (CACM)* 59 (2016), 78–87. http://dl.acm.org/citation.cfm?id=2854146

[20] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. 2012. Contemporary Peer Review in Action: Lessons from Open Source Development. *IEEE Software* 29, 6 (2012), 56–61. https://doi.org/10.1109/MS.2012.24

[21] Peter C. Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 202–212. https://doi.org/10.1145/2491411.2491444

[22] Adam Roberts, Hyung Won Chung, Gaurav Mishra, Anselm Levskaya, James Bradbury, Daniel Andor, Sharan Narang, Brian Lester, Colin Gaffney, Afroz Mohiuddin, et al. 2023. Scaling up models and data with t5x and seqio. *Journal of Machine Learning Research* 24, 377 (2023), 1–8.

[23] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 181–190.

[24] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. Autotransform: Automated code transformation to support modern code review process. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 237–248.

[25] Rosalia Tufano, Ozren Dabić, Antonio Mastropaolo, Matteo Ciniselli, and Gabriele Bavota. 2024. Code Review Automation: Strengths and Weaknesses of the State of the Art. *IEEE Transactions on Software Engineering (TSE)* (2024).

[26] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2291–2302.

[27] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.

[28] T. Winters, T. Manshreck, and H. Wright. 2020. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media. https://books.google.ch/books?id=TyIrywEACAAJ

Received 2024-04-05; accepted 2024-05-04