Leveraging Test Plan Quality to Improve Code Review Efficacy

Lawrence Chen
Meta Platforms, Inc.
USA

Rui Abreu
Meta Platforms, Inc.
USA

Tobi Akomolede
Meta Platforms, Inc.
USA

Peter Rigby*
Concordia University
Canada

Nachiappan Nagappan
Meta Platforms, Inc.
USA

Satish Chandra
Meta Platforms, Inc.
USA

ABSTRACT

In modern code reviews, many artifacts play roles in knowledge-sharing and documentation: summaries, test plans, and comments, etc. Improving developer tools and facilitating better code reviews require an understanding of the quality of pull requests and their artifacts. This is difficult to measure, however, because they are often free-form natural language and unstructured text data. In this paper, we focus on measuring the quality of test plans at Meta. Test plans are used as a communication mechanism between the author of a pull request and its reviewers, serving as walkthroughs to help confirm that the changed code is behaving as expected. We collected developer opinions on over 650 test plans from more than 500 Meta developers, then introduced a transformer-based model to leverage the success of natural language processing (NLP) techniques in the code review domain. In our study, we show that the learned model is able to capture the sentiment of developers and reflect a correlation of test plan quality with review engagement and reversions: compared to a decision tree model, our proposed transformer-based model achieves a 7% higher F1-score. Finally, we present a case study of how such a metric may be useful in experiments to inform improvements in developer tools and experiences.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; **Acceptance testing**; **Walkthroughs**.

KEYWORDS

Code Reviews; Pull Requests; Test Plans; Natural Language Processing.

ACM Reference Format:
Lawrence Chen, Rui Abreu, Tobi Akomolede, Peter Rigby, Nachiappan Nagappan, and Satish Chandra. 2022. Leveraging Test Plan Quality to Improve Code Review Efficacy. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3540250.3558952

*Work performed while on sabbatical at Meta Platforms, Inc.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558952

1 INTRODUCTION

At Meta, the *test plan* is a component of our code review process and culture. In addition to the automated test suites that are run for each code change, developers also provide free-form documentation summarizing the testing process employed to verify the code changes. The goal is to describe the manual steps the developer took during the development process to sanity check the code changes, such as commands to run specific automation, screenshots of expected UI changes, or steps to conduct a full end-to-end test.

These test plans are similar to the ones found in the software quality assurance process or the test plans described in the IEEE Standard for Software Test Documentation [4]. However, Meta's test plans are generally concerned with the scope of one set of code changes at a time, rather than the entire software product.

Hence, test plans are complementary to other forms of validation. For example, in code review, the formal compilable source code is studied to find defects; in testing, regressions are identified by test failures. In contrast, test plans serve as simple written walkthroughs of the system behavior that provide assurances and context to reviewers. Test plans are highly effective at Meta, and one of our goals is to describe test plans to the larger SE community.

The flexible and lightweight nature of test plans make them easy to write and easy to understand, but this makes measuring the quality of test plans significantly more difficult. Our goal is to develop a model that can differentiate good test plans from bad, which helps us better understand the overall state of test plans at Meta. This information could also be used to help suggest to developers when they may need to improve the quality of test plans to better improve the code review process.

In particular, this experience paper aims to gain an understanding of the following research questions:

(1) **Modeling Quality:** Can we use a data-driven approach to model holistic test plan quality that aligns with the opinions of the developers?
(2) **Applying NLP Techniques:** Do the state-of-the-art transformer-based models used in NLP tasks translate to the code review domain and outperform more trivial methods that require feature engineering?
(3) **Correlation With Review Engagement:** Is our data-driven test plan classifier correlated with reviewer engagement?
(4) **Correlation With Regressions:** Do pull requests that get reverted or are associated with outages have test plans with lower quality?

To help us answer these questions, we first need to understand how test plans are used at Meta (for details, see Section 2.3), but, at

a high level, test plans at Meta are used to augment the code review process, acting both as a procedure to exercise functionality and as part of the documentation. Yet another fundamental characteristic of a test plan is that it should be repeatable: the contract is that executing the specified steps should be sufficient to have confidence that the feature works correctly.

We have some intuition of data that can be mined from test plans, such as counting words, attached media, or links, and we could turn these into metrics. These generally make sense as desirable features to have in a test plan and could indirectly signal the quality of the test plan. However, they are not easily consolidated into a single holistic quality metric, and any trivial heuristic is likely neither flexible nor robust.

But, most importantly, these trivial extractions ignore the most valuable data in a test plan, which is the free-form natural language text. Developers often use test plans to detail replication steps or end-to-end integration steps. The conciseness and clarity of the natural language descriptions could be informative of a test plan's quality, as might interactions between the text and attached media or links. Therefore, we look to data-driven natural language processing (NLP) techniques to leverage such information.

We define the problem as a binary classification model, with labeled test plans professionally rated as *GOOD* or *BAD* by 500 Meta developers. We leverage a pre-trained RoBERTa language model [15] for our architecture, which takes the test plan as a natural language input. We tokenize the test plan string and generate an embedding, using that model, then utilize the few-shot learning Matching Network architecture [10] to classify the test plan embedding as either *GOOD* or *BAD*.

We observed that the natural language approach, via fine-tuning a RoBERTa-based model, outperformed our decision tree baseline and improved the F1 score by 7%. This suggests that the general pre-trained RoBERTa model is successful at learning the patterns of test plan quality, simply through fine-tuning on this "code review"-specific domain. By capturing both structural and semantic information, our transformer-based approach outperforms the baseline without the need of manual feature engineering for its input features.

To summarize, this paper makes the following contributions:

- We define test plans at Meta and how they are used to augment the code review process.
- We demonstrate that NLP architectures, such as a RoBERTa-based model, can be applied to the code review domain to quantify the quality of test plans.
- We evaluate our approach on the task of measuring the quality of test plans and demonstrate how such a quality heuristic may be used to inform improvements in developer tools.

The main take-away message of this paper is that:

> Pull requests with high quality test plans are observed to: 👍 be involved in fewer outages, 👍 be reverted fewer times, and 👆 have more reviewer engagement.

The remainder of this paper is structured as follows: in Section 2, we describe how code review is done at Meta, discuss what test plans are in the context of code review at Meta, and introduce our internal code review engagement metric. In Section 3, we introduce our NLP-based holistic approach to modeling test plan quality. We also introduce a baseline technique based on decision trees. In Section 4, we present our results and discuss threats to validity. In Sections 5 and 6, we discuss our findings in the context of related work and describe potential applications to downstream tasks. In Section 7, we conclude the paper.

2 BACKGROUND

This section details the Meta code review process, introduces a custom code review engagement metric, and discusses what test plans are in that context. It is noteworthy that test plans are a communication channel between the author and the reviewers, not a *proof* that the changes were tested. One common workflow is allowing a reviewer to try out the modification in the pull request before it gets committed.

2.1 Code Review Workflow at Meta

`Phabricator`, an open-source project, is the backbone of Meta 's Continuous Integration system[1], and is the surface for the modern code review process. Developers use `Phabricator` both to submit pull requests and to comment on others' requests before they are accepted into the codebase (or are discarded). More than 100,000 pull requests are committed to the central repository every week at Meta, using `Phabricator` as a central gate-keeping, reporting, curating, and testing system [9].

The author uploads the changed code to `Phabricator`, includes a *test plan*, assigns reviewers (or groups of reviewers) and "publishes" the pull request, . The act of publishing a pull request sets its status to "needs review", making it visible to all assigned reviewers. At least one reviewer's approval is required to accept (and, therefore, ship) a pull requests.

`Phabricator`'s UI displays the contents of the pull request (see Figure 1): a title, summary, the code changes, and the test plan; detailed in Section 2.3. The reviewer can add comments (which are visible to everyone), accept, send back to the author (requiring further changes), resign, or commandeer (becoming the author).

The author of the pull request, meanwhile, has several available actions. They can amend their code change and update the pull request to a new iteration, request another review pass, add comments (to, e.g., explain an update or address reviewer feedback), pause the review process until all changes are complete, abandon the request, or (once the request is accepted) ship it to production. Shipping a code change is gated on approval by reviewers.

2.2 Code Review Quality and Engagement

One research question explored in this paper involves tying our holistic test plan classifier to downstream metrics of review quality and engagement. To quantify review quality and engagement, we developed a custom heuristic that is a function of the size of the changes, the number of review comments, and the time spent reviewing.

Code review teams at Meta utilize several review commentary and cycle characteristic metrics, which are then combined into

[1] http://phabricator.org

Figure 1: An example view of a pull request under review in `Phabricator`. Authors and reviewers can interact via the pull request review page in `Phabricator`. The review has several sections: the summary, changes, reviewers, interactions between the reviewers and author, test case information and status, results of static analysis, historical information, etc.

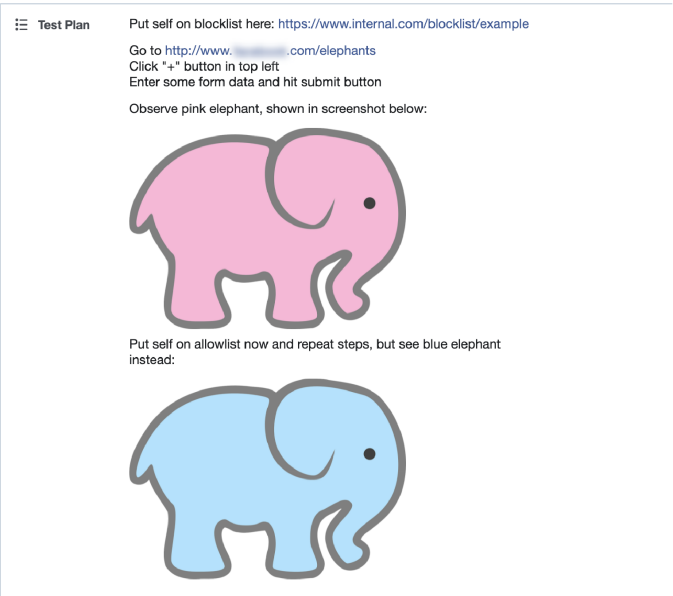

Figure 2: A simplified example test plan for a diff that implements a blocked list of users. The test plan details the set of steps and UI outcomes that will verify that the code works as expected.

a single engagement metric to conveniently monitor the level of engagement that pull requests receive. This is also interpreted as a proxy for the quality of the review at Meta, as it indicates how developers interact during a "pull request review", which we define to be the process of inspection and discussion undertaken by the reviewers, subscribers, and author of a pull request. Moreover, we limit discussion to mean the set of review comments posted within the Meta's code review tool, `Phabricator`. This definition excludes any review discussion that might occur by any other means, such as in-person, direct messaging, or over videoconferencing.

In the context of this paper, this exact definition is not crucial to our work. We treat this pull request engagement metric as a black box (an interested reader can define her own quality and engagement heuristic to replicate our study in a different company/context), and the main idea is that this metric is used at Meta to measure and represent the quality and engagement of the pull request review.

We will outline the set of features that are input to this supervised, machine learning-based engagement metric, which comes from two main dimensions: review commentary and cycle characteristics. Regarding review commentaries, the set of features are: number of substantive comments[2], number of substantive comment threads, number of substantive head-level (non-reply) author comments, and the number of reviewers that leave substantive comments. With respect to cycle characteristics, the set of features are: number of pull request versions and number of times the pull request was set to "Needs Revision" status before landing.

The key takeaway of using an engagement metric is that it measures the activity and interaction between the reviewer and the authors. More engagement and activity logically lead to better reviews, and thus to generally fewer reversions after shipping code. We validated our engagement metric, and found that this was indeed the case: our internal analysis of this engagement metric indicates that pull requests that have low engagement scores are 6.6× more likely to be reverted than pull requests with high engagement scores. Hence, we use this engagement metric to measure the

[2]A substantive comment is defined as a comment that is considered nontrivial. We filter out trivial comments that are not engaging. This is a somewhat loose definition, but includes simple comments of affirmation, such as "good job" or "nice one".

review quality of a pull request, and in later sections we demonstrate that better test plans, as measured by our proposed technique, result in better review quality and less pull request reversions.

2.3 What Are Test Plans?

As mentioned before, a test plan is a repeatable list of steps which documents what the author has done to verify the behavior of a change, and is a required component of the code review process at Meta. A good test plan convinces a reviewer that the author has been thorough in making sure the change works as intended and has enough detail to allow someone unfamiliar with the code change to verify its behavior. It is worth noting that test plans are not meant to replace automated testing mechanisms that may be added as part of the pull request (e.g., unit testing or UI testing). Instead, test plans are complementary to automated test plans in the sense that they are intended as an aid to the reviewer of the pull request in replicating the functionality that is being implemented. Hence, a test plan is a (mostly textual) step-by-step guide to anyone trying to figure out the impact of the modifications in the pull request: the clear the steps, the more easily developers can execute it.

We observe that test plans at Meta are used to:

- Allow reviewers to try out features before they are committed.
- Allow reviewers to identify edge-case user behaviors to consider in an end-to-end test plan.
- Document clear steps for consistent reproduction or testing if the pull request is updated during the review process.
- Document testing procedure for future engineers to verify what the expected behavior of the code should be.

Figure 2 is a simplified example of a test plan for a change in UI. The plan contains links to verify that the changes are as expected. Screenshots demonstrating the before and after make it clear to the reviewers what the intended effects are, rather than only relying on unit tests, which may be less intuitive to check UI changes during code review. Moreover, writing UI test scripts can be time consuming and difficult to maintain, as well as UI scripts are inherently fragile [27].

A non-UI example could be for new API endpoints, in which test plans can demonstrate example calls to the new endpoints and detail the expected inputs and outputs after the code changes. This makes it easier for reviewers to quickly confirm that they agree with the current behavior of a new feature on an API level, rather than having to parse test cases, which may be less readable.

It is important to note that test plans are used not just for functional changes; pull requests with non-functional changes also benefit from high quality test plans. Moreover, note that test plans are not meant to replace any type of testing, but rather to explain to reviewers how you know your code is behaving as expected. Test plans enhance the code review process, serving as additional readable documentation, along with more rigorous unit tests that are also enforced at Meta.

The contract of a test plan is that if a developer were to replicate the steps of a test plan, the developer should be confident that the code changes work as expected. This means not just testing the new code, but also making sure that nothing else will be broken. In an ideal world, there should be tests protecting the rest of the code, but it is also the responsibility of the pull request's author to make sure that the pull request is not introducing any regressions. Test plans allow authors to easily demonstrate that with more clarity and precision. For instance, if code is to be deleted in a pull request, the test plan should include a change impact analysis explaining how the developer knows this code is not being used — e.g., using test coverage or code search. In general, a good test plan should give clear, concise, and reproducible instructions that someone else can easily follow.

3 OUR TEST PLAN CLASSIFIER APPROACH

We approached the holistic test plan classifier by modeling it using NLP techniques. The current state-of-the-art for NLP problems, such as text classification [13], machine translation [17], and text generation [11], typically involve the now ubiquitous transformer based architecture [25, 28, 30, 34], and we are interested to see if such success can transfer over to our code review domain-specific use case. Specifically, we used the RoBERTa transformer architecture [15] to model test plan quality. Aside from a few basic preprocessing steps, such as using keywords to represent screenshots or links, we were able to use the test plan string as the raw input data, like natural language data.

RoBERTa [15], short for *Robustly Optimized BERT Pre-training Approach*, is a pretrained natural language processing system that improves on Bidirectional Encoder Representations from Transformers, or BERT, the self-supervised method released by Google in 2018 [7]. BERT achieved state-of-the-art results on a range of NLP tasks while relying on unannotated text drawn from the web, as opposed to a language corpus that has been labeled specifically for a given task. The technique has since become popular both as an NLP research baseline and as a pre-trained model backbone for downstream language tasks. For RoBERTa [15], the objective was to optimize the training of BERT architecture in order to take less time during pretraining. RoBERTa has been shown to produce state-of-the-art results — including the impact of training data and training time — on the widely used NLP benchmarks, such as GLue [33], as well as SuperGLUE [32], and SQuAD (data-constrained setting) [22, 23].

Our model architecture consists of a RoBERTa model which produces embeddings of length 1024 for our test plan text input[3], which is then fed into a simple fully-connected layer for classification. We started with an open-source RoBERTa model from Hugging Face that is pretrained on general NLP tasks,[4] and then fine-tuned the whole model end-to-end on our test plan data. We used the default pretrained tokenizer, adding no additional custom tokens.

Our only data preprocessing involved replacing links or screenshots that may be arbitrarily long with shorter standardized keywords. Specifically, all multi-line code markup sections (often representing large log outputs) were replaced with "<codeblock>", url links were replaced with "<url>", and attached images or other media were replaced with "<screenshot>". Note that these keywords were just string replacements, not explicitly added as special

[3] Test plans that are longer than 1024 tokens are truncated. Note, however, that most test plans in our dataset fit within this limit.
[4] https://huggingface.co/transformers/model_doc/roberta.html

tokens to the vocabulary. The chosen keywords were arbitrary, and only meant to clean and shorten input data with standardized representations.

In this supervised learning modeling, we also need labels for our data. Our labels were *GOOD* or *BAD* quality ratings that we gathered from experienced software developers at Meta. In total, we collected over 650 data samples of test plans and their corresponding ratings as labeled by about 500 Meta engineers (see Section 4.1 for more details).

Our goal with the model is to learn a model representation of what reviewers at Meta deem to be *GOOD* quality test plans. Essentially, this task is similar to a sentiment analysis NLP task (e.g., [6, 20, 24]), but specifically for Meta 's code review domain. By using a pre-trained RoBERTa model, we were able to leverage the natural language representations learned from general NLP tasks and apply them to test plans via transfer learning.

In addition, we experimented with replacing the fully-connected layer in the RoBERTa classifier with a Matching Network wrapper [31] to take advantage of "one-shot learning" principles, which further improved performance. Matching Networks, which are designed to work well with little data, work by utilizing a "support set". In a high level description, Matching Networks are a deep learning "k-nearest neighbors" [21], in which the model learns to embed inputs with the context of a *support set* acting as the "neighbors". Our approach is to use this contextual embedding technique on top of the RoBERTa embeddings. We argue that this will allow us to maintain the RoBERTa classifier's strength of flexibility and performance, while addressing its potential weaknesses of overfitting smaller datasets. Additionally, one potential advantage that we get from using a Matching Network architecture is that it provides a lightweight notion of interpretability. The Matching Network architecture compares the input test plan to a support pair of *GOOD* and *BAD* labeled test plans, and the output of the model effectively indicates which example test plan in the support pair is most similar to the input test plan in question. This sort of interface may be more intuitive to a human user than a simple binary label output. However, incorporating a more fine-grain notion of model explanation is left as future work.

For this extension on the RoBERTa model, we simply just re-implemented the Matching Network architecture as proposed in the paper [31]. But instead of deriving the input features to the Matching Network from a convolutional neural network (CNN) [1], as in the original paper did for image processing, our inputs are the RoBERTa model embeddings generated from our test plan data. The whole model is then trained end-to-end. The Matching Network architecture uses an example "support set" data, allowing it to learn the similarities between new data samples and the support set data. Multiple inference passes can be done with the Matching Network architecture by utilizing different pairs of support data, allowing this model to effectively behave as an ensemble model as well, further improving performance.

We do not have any models or heuristics as the *status quo* that we can use as our baseline comparison, so we also developed a simple decision tree model that takes the feature-engineered metrics as input, as listed in Table 1. This model is not used in practice, but rather it is a representation of a naive solution which we use to compare with our new proposed technique. It is an example of how one might come up with manually hand-crafted heuristics or features to model test plan quality, which we argue is less scalable and less performant than data-driven approaches that we explore. As for the features listed in Table 1, not that it is not always the case that the more, the better. As an example, having many URLs with no text to explain it may be an indication of a bad test plan. This list is a good representation of the type of manually defined metrics the code review team may have used as their main signal of test plan quality prior to our work. The decision tree, therefore, represents what a possible simple heuristic unifying the metrics might look like for a holistic model classifier. The key difference for this baseline is that it requires manual feature engineering and heuristic definition, as opposed to our RoBERTa-based models that require little data preprocessing. This decision tree baseline also fails to capture syntactic structure or semantic information.

Table 1: List of manually constructed features for the Decision Tree baseline model.

Feature Name	Description
Non-code Length	Length of test plan, excluding sections formatted as multi-line code markup
Num URL	Number of url links included in test plan
Has Codeblock	Whether test plan contains sections formatted as multi-line code markup
Has Codeline	Whether test plan contains single-line code markup (typically representing run commands)
Has Test Command	Whether test plan contains common test commands, e.g. hack unit tests, jest, etc.
Has Screenshot	Whether test plan contains images, videos, or other media attachments (which are typically screenshots or screen recordings)
Has Common Commands	Whether test plan contains common run commands, such as for linters, formatters, or static analyzers

Figure 3 shows the three approaches considered in our experiments. All in all, our proposed models have the benefit of

- no feature engineering (other than tokenization / simple regex replacement for media and links);
- capturing natural language data, including both syntactic and semantic information;
- being more generalizable without relying on rigid rules.

4 EMPIRICAL EVALUATION

This section details the empirical evaluation of our test plan classification approaches. The industrial context of the empirical evaluation is Meta's continuous integration system.

4.1 Setup

To obtain test plan labels, we identified a list of 500 developers who have conducted the most code reviews in the past year. For this list of top reviewers, we randomly sent them a survey during the code review process asking them to rate the test plan quality as either *GOOD* or *BAD*. Using this method, we collected over 650

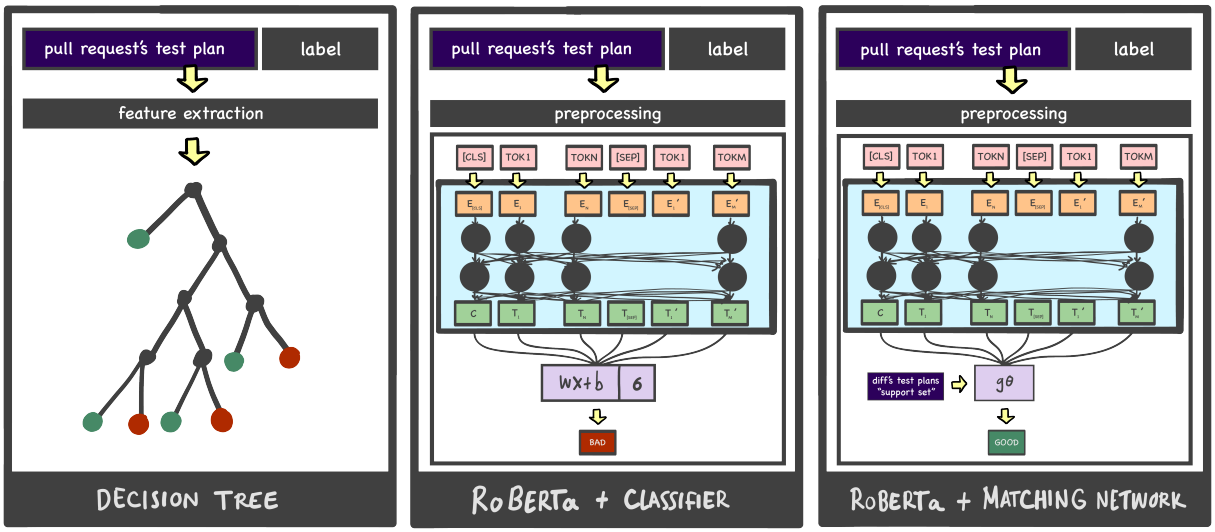

Figure 3: Approaches considered in our study.

human-rated test plans as this fold our training set. As we wanted to gauge the opinions of the reviewers in a realistic setting, each test plan was rated by the reviewer (often only one) of the pull request using `Phabricator`.

For training each of our three models, we employed a 65-15-20 data split for training, validation, and testing. Note that the support set used during the matching network training and validation is sampled from their respective datasets. During the inference or testing phase for matching network, we use a separate support set that we set aside, consisting of 3 support pairs.

Following the questions outlined in the Introduction, concretely, we look to answer four research questions in our empirical experiments:

(1) Can we use a data-driven approach to model holistic test plan quality that aligns with developer opinions?
(2) Do state-of-the-art transformer-based models used in NLP tasks translate to the code review domain, and outperform more trivial methods that require feature engineering?
(3) Is our data-driven test plan classifier correlated with reviewer engagement?
(4) Do pull requests that get reverted or are associated with outages have test plans with lower quality?

For the first two research questions, we simply evaluate our model performance on the test set, using the F1-score as our metric. The F1-score is a widely used metric for binary classification, which makes it a suitable and simple way to compare the models with which we are experimenting. Formally, the F1-score is the harmonic mean of precision and recall:

$$\text{F1-score} = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

where *Precision* is the number of true positive results divided by the number of all positive predictions, including those not identified correctly, and *Recall* is the number of true positive results divided by the number of all samples that should have been identified as positive.

The latter two research questions are concerned with tying our data-driven quality metric with some related downstream ground-truth metric. To do so, we will be applying our data-driven model on unlabeled data to measure their test plan quality, then do simple metric comparisons to find correlations. The third question is interested in how test plan quality correlates with review quality and engagement, as discussed in section 2.2. The final question is concerned with test plan quality for pull requests that are related to some form of regression, and thus intuitively of lesser quality.

4.2 Results

We measure the model performance by measuring the F1-Score of the predictions on our test set (see Table 2). We observed that the RoBERTa Simple Classifier model and RoBERTa Matching Network model improved the average F1-score by about 4% and 7%, respectively, compared to the Decision Tree baseline model.

Our observations suggest that our model is able to capture and model the sentiment of Meta developers towards sample test plans with about 90% accuracy. This is quite substantial, considering developer ratings of test plan quality is, to a certain extent, a subjective measure.

> RoBERTa Simple Classifier model and RoBERTa Matching Network model improved F1-score by about 4% and 7% on average, respectively, compared to the Decision Tree baseline model.

We also observe that our predicted test plan quality is correlated with a pull request's review quality (or review engagement), as shown in Table 3. First, we predicted the quality of unlabeled test plans using our Matching Network model. When controlled for lines of code (because pull requests of different sizes may have different levels of review engagement), pull requests with predicted *GOOD* test plans have on average 6.6% higher reviewer engagement. If we look specifically at the moderately sized pull requests, which avoids confounding variables of trivial or overly-complex code changes, we see that pull requests with *GOOD* predicted test plans have 10.5% higher reviewer engagement, a statistically significant increase. This suggests that our test plan classifier does not just align with our developer sentiment, but can also be tied to concrete metrics that demonstrate the impact of test plan quality.

> Pull requests with predicted *GOOD* test plans have on average 6.6% higher reviewer engagement; this value increases to 10.4% higher reviewer engagement if only considering pull requests with moderately sized changes.

Finally, we also gathered a dataset of pull requests associated with known outages at Meta: in particular, we collected an order of 10^4 diffs associated with reversions and half as many associated with outages. These pull requests may not necessarily be the direct cause of these outages, but are mentioned in outage reports which strongly suggests they may have caused regressions. Moreover, we also gather a dataset of reverted pull requests.

When we run the inference on the test plans of these pull requests, we see that their predicted quality is noticeably lower. As shown in Table 4, for pull requests associated with outages, we see that the average test plan quality rating is 3.8% lower than the rating for other pull requests. However, we observed that this difference is not statistically significant. We argue that a potential reason for this observation is that we have a very limited amount of sample outage-related pull requests as there are not that many outages at Meta.

When we look at reverted pull requests, which is a larger sample size, we see that the predicted test plan quality is 8.9% lower than the ratings for other pull requests. With the larger dataset, we see that this difference is indeed statistically significant. By using our model, we can quantify that lower test plan quality is correlated with poorer quality pull requests that cause regressions or are eventually reverted, matching our intuition that test plan quality affects the overall pull request and code review quality.

> The average test plan quality of pull requests associated with outages is 3.8% lower than the average test plan quality rating for other pull requests; The average test plan quality associated with reverted pull requests is 8.9% lower than the average test plan quality rating for other pull requests.

4.3 Threats to Validity

This subsection discusses potential external, construct, and internal threats.

Table 2: Model performances on test set, predicting *GOOD* or *BAD* labels for test plans.

	GOOD Label F1-Score	BAD Label F1-Score	Average F1-Score
Decision Tree baseline	0.779	0.877	0.828
RoBERTa + Simple Classifier	0.841	0.893	0.867
RoBERTa + Matching Network	**0.871**	**0.921**	**0.896**

Table 3: Review engagement metric comparison for pull requests with *GOOD* vs. *BAD* (control) test plans, as predicted by trained RoBERTa + Matching Network model and controlled by lines of code of pull request.

Lines of Code	Avg. Metric Change	T-Value	P-Value
Bottom Third Quantile (<18 Lines)	+2.7%	0.984	0.325
Middle Third Quantile (>18 Lines and <73 Lines)	+10.5%	34.370	1.26×10^{-256}
Top Third Quantile (>73 Lines)	+7.4%	25.332	5.35×10^{-141}

Table 4: Test plan classification as predicted by trained Matching Network model, comparing metric change for outage-related or reverted pull requests with other pull requests.

	Avg. *GOOD* Test Plans	% Change	T-Value	P-Value
Other Pull Requests	0.235	–	–	–
Pull Requests Associated with Outages	0.226	-3.8%	-0.936	0.350
Reverted Pull Requests	0.214	-8.9%	-3.647	0.000267

4.3.1 External Validity / Generalizability. Drawing general conclusions from empirical studies in software engineering is difficult, because any process depends, to a degree, on a potentially large number of relevant context variables. While this analysis was performed at Meta, it is possible such results might not hold in other environments/domains/companies. As an example, the results might not hold for small and/or local teams, where the value of explicitly written test plans may not bring value over in-person communications. For this reason, we cannot assume *a priori* that the results of a study generalize beyond the specific environment

in which it was conducted. Researchers become more confident in a theory when similar findings emerge in different contexts [2]. Therefore, we urge other researchers to replicate a similar study in other environments.

4.3.2 Construct Validity. Construct validity is used to determine whether the dependent and independent variables accurately represent the concepts they are supposed to measure. A particular threat to construct validity of our work is that we leverage an in-house metric to review quality (review engagement) that has otherwise not been validated by the scientific community. To mitigate this threat, we have conducted an extensive study with the metric at Meta; hence, the metric is validated by developers in a real context.

Other threats to construct validity are related to the suitability of our evaluation metrics and the quality of the labeled datasets that we use. F1-score is widely used to evaluate solutions, such as our approach. As for labeling the datasets, there is a possible confounding variable that teams who spend more time writing good test plans have a culture that also separately results in them writing better pull requests / reviews.

Moreover, the quality labels of test plans that indeed have an element of subjectivity. However, given that the proposed approach is to be used by humans, we argue that this subjectivity is acceptable. In particular, we wanted to see if a model could capture the general trend or opinions of human opinions, even if subjective. We then found that our model was decent at modeling human subjective opinions, and even more interestingly, they have significant correlations with objective metrics, like review engagement, reversions, and outages.

Yet another potential threat to the validity regarding the labels of the test plans is that one might question whether a binary classification is a good representation of test plans. One might argue that there may be more levels to the quality of the plan other than simply *GOOD* or *BAD*.

4.3.3 Internal Validity. One potential threat to internal validity relates to errors that we may have made in our experimental setup/pipeline. This threat has been mitigated by careful peer-review of the pipeline by the authors.

Moreover, as mentioned before, we gathered a dataset of pull requests associated with known outages at Meta. However, these pull requests may not necessarily be the direct cause of outages; they are mentioned merely in outage reports. Although strongly suggesting that they may have caused regressions, outage-associated pull requests are not necessarily guaranteed to be the root causes of outages. Additionally, note that the size of the dataset is limited.

We did not perform any hyperparameter tuning for our models, which leaves open the possibility that these models can be further improved through a rigorous hyperparameter optimization. Our decision tree baseline was trained with a depth threshold of 4, the criterion set to "entropy", min_samples_leaf set to 4, and all other hyperparameters set to default in the sklearn API[5] [29]. For our gradient-based models, we trained both for 50 epochs with batch sizes of 16. We set the weight decay to 0 and the learning rate to 1×10^{-5}. These hyperparameter decisions were largely driven by the motivation to keep resource usage manageable and reasonable. By using non-optimized defaults, the hyperparameters were fixed before looking at the test set to ensure no contamination.

[5]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

5 DISCUSSION: DOWNSTREAM USAGES

One strong implication of our findings is that higher quality test plans will lead to better review engagement. That is, a bad test plan will make it difficult for reviewers to properly review or test the code, whereas a good test plan means reviewers can more efficiently catch issues or provide reviewers more context and confidence, prompting actual productive conversations or reviews to happen.

We explain below usage scenarios for external researchers and engineers to leverage such a system motivated by usage at Meta.

(i) Use it to surface high-quality test plans as recommendations to developers. At Meta, we have internal tools that help developers find test plans from related code changes and files. We can use the proposed model to filter or rank higher quality test plans when developers request test plan examples. This can encourage higher quality test plans at Meta, and also improve the developer and code review experience. Moreover, surfacing high-quality test plans may serve well as an educational tool for novice developers.

(ii) The second is to use it as a distinctive feature in other prediction modeling approaches. As an example, `Phabricator` offers a functionality to predict the time it will take to review a pull request. One can imagine building a model that considers the test plan classifier as one of its input features. We argue that better test plans will help with the accuracy of such predictions.

(iii) To use it as a downstream metric in our experiments. Returning back to our original motivation, we would like to make sure that new features or code review efforts do not hurt the quality of our pull requests. In our experiments, we can use our classifier to make sure that the quality of test plans does not deteriorate or maybe we can even find ways to improve it.

A example case study is using Test Plan Linters. At Meta, we developed some "linters" for test plans, hoping to nudge reviewers to create higher quality test plans. For example, to ensure that the test plans provide as much context as possible, we developed an "Attach Screenshot" linter and "Missing Link" linter.

The "Attach Screenshot" linter predicts when a test plan should contain a screenshot based on historical data, and recommends developers attach an image if they are missing one. The "Missing Link" linter asks the developer to also provide an URL link to document the source of the attached media if it is not included.

In our experiment, we want to test if these linters would indeed improve test plan quality and thus reduce other downstream metrics we care about, or perhaps these linters were ignored, or, even worse, resulted in developers doing the bare minimum instead.

What we found was that pull request staleness had decreased and test plan quality, as measured by our model, had increased. This is a great empirical anecdote of how test plan quality is tied closely with other important code review metrics, and can help us understand the impacts of new features in our experiments.

6 RELATED WORK

There are numerous studies in academia investigating how developers perceive review quality. In "Code Review Quality: How Developers See It" [12], the authors surveyed developers at Mozilla

and found that key aspects of a well-done code review include thorough feedback, advice around correctness issues, and the quality of the code changes themselves. In "Characteristics of Useful Code Reviews: An Empirical Study at Microsoft" [3], the authors surveyed developers to build a set of criteria with which to label code review comments as "useful" or "not useful". They then employ this criterium to manually label a training set with which they train a classifier that detects the usefulness of a comment.

It should be noted that the above studies look at human-based, subjective measures of quality. The former study directly surveys developers, whereas the latter builds a classifier around criteria formed from human sentiment. The next series of studies mentioned will look at code review quality from the perspective of objective measures of quality, such as the incident of post-release defects or occurrence of anti-patterns.

McIntosh *et al.* [18] studied whether various code review participation metrics have an effect on post-release defects, in the context of modern, asynchronous code reviews. Rather than attributing defects to pull requests directly, the authors attribute defects to "components", which are folder-level aggregations of files. Post-release defects are identified by searching for version control commit messages that contain keywords like "bug", "fix", "detect", or "patch". Metrics on review characteristics and human factors (such as the number of authors that touch a component) are calculated for the six-month period prior to a product release, rather than individual pull requests. They find that component reviews without discussions and reviews with lower reviewer participation rates (i.e., self-approved reviews) tend to have higher post-release defect counts.

A replication of the the above McIntosh *et al.* study on the Google Chrome project is discussed in another work [14]. This study again associates post-release defects with files at the directory level, rather than individual patches themselves. Post-release defects are identified by scraping the Chrome issue tracker, searching for issues submitted in the time period after the current release. They find that review participation measures have an inconsistent explanatory power across projects and releases, leading them to model the problem with a Bayesian Network-based approach [8, 35]. The Bayesian Network methodology showed that review participation measures have an indirect effect on post-release defects, with prior defects, component size, and the number of inexperienced authors contributing the most to the incident of post-release defects.

Mäntylä and Lassenius studied whether code review finds issues related to the functional correctness of code or "evolvability issues", such as the structure or documentation of source code [16]. They give a taxonomy of software defects, and find that 71.1% of the findings in industrial code reviews are related to evolvability defects, with 21.4% of the findings corresponding to functional defects and 7.5% being false positives. It should be noted that the code review methodology used in this study was a synchronous, in-person, and recorded meeting, in which multiple reviewers commented on code changes in real time. In the industrial setting, nine review sessions were observed in total. The author of the study observed code review sessions and manually identified defects as being related to either functional or evolvability issues. Thus, defects in this study are attributed to what reviewers identify before the changes are shipped. It should also be noted that this review methodology is starkly different from the Meta 's asynchronous code review practices.

McIntosh *et al.* have further expanded on their previous study on the relationship between code review and software quality, this time specifically trying to determine whether code reviews inhibit the occurrence of "anti-patterns" in code [19]. As in the previous McIntosh *et al.* study, anti-pattern defects are attributed to components, or directory-level file sets. Seven anti-pattern types are detected using automated tooling. They find that code review coverage and participation reduce this incident of anti-patterns in software components.

There is a work [5] that makes the strongest claims as to the value of code review in relation to finding software defects. The authors state that only 15% of the code reviewer comments indicate a possible defect. The usefulness of the review comment, as judged by the author, is correlated with the experience of the reviewer, with more experienced reviewers giving more useful feedback. They find that review usefulness decreases with the size of the changed file set, with a noticeable drop-off occurring around 20 changed files.

"Review Participation in Modern Code Review" [26] examines what factors lead to faster reviewer engagement, rather than what aspects of reviewer engagement impact software quality. It is worth mentioning because its results align with those of the above studies around the relationship between reviewer characteristics and code reviews. In this study, not all pull requests that are shipped receive code review, and thus there is a question as to why some pull requests receive review and others do not. This study finds that the greater the number of prior pull requests reviewed by an assigned reviewer, the more likely the pull request will actually receive review. This aligns with previous studies, which find that reviewer characteristics, such as the amount of previous contributions to the code components, the amount of code reviews performed in the past, and social factors, such as a reviewer's personality or standing in the team, impact the quality of code review and the resulting software.

To summarize, the prior literature in this space indicates that it is an unanswered question whether review characteristics impact software quality. Existing studies find that other characteristics, such as those relating to the reviewer or the code quality itself, can have more explanatory power on whether post-release software defects occur (so much so that these other characteristics are directly controlled for in many of these studies). To put it plainly, it is still unproven that code review reliably catches bugs. If the vast majority of code review is truly concerned with evolvability and stylistic issues, as suggested by the above studies, an argument could be constructed that it would be wiser to invest more in automated code quality enforcement tooling, rather than better code review practices. Before we get too carried away with this, it is important to make some very clear distinctions between the code review practices examined in this prior literature and those employed at Meta, as well as the underlying questions being investigated.

We do argue — and our study suggests —- that including a high-quality test plan to a pull request will lead to better code quality overall. In this study, we are concerned with the quality of the test plan in a pull request. Our question is, what characteristics of a given pull request, if any, reliably predict whether it is likely to be a "blame pull request". To refresh the reader's mind, a blame pull

request is one that is implicated in an outage or otherwise reverted for any other reason.

7 CONCLUSIONS AND FUTURE WORK

As part of the effort to measure and improve pull request quality, in this industrial experience paper, we describe our efforts in devising a classifier for test plans. We started from the hypothesis that increased test plan quality could help improve review quality and engagement. Our goal is to demonstrate that data-driven approaches could be leveraged as a meaningful heuristic. Note that we are not attempting to set a new benchmark with high evaluation accuracy; instead, we are showing that out-of-the-box language models make for good heuristics that align with both human subjective opinions and objective metrics.

We frame the problem of quantifying quality of a test plan as a binary classification model. Leveraging the success of NLP-based techniques in other domains, we propose an approach based on a pre-trained RoBERTa language model which takes in the test plan as a natural language input. The model was fine-tuned using opinions of about 500 developers on over 650 test plans. Our empirical evaluation suggests that the proposed approach is able to capture developer sentiment and reflect a correlation of test plan quality with review quality/engagement, outages, and reversions. Compared to a decision tree model —the baseline model—, our proposed transformer-based model achieves a 7% higher F1-score, i.e., the proposed approach is more accurate than the baseline and produced statistically significant correlations with the objective metrics. We have also presented a case study of how such a metric may be useful in experiments to inform improvements in developer tools and experiences.

We have discussed potential downstream applications in a previous section, paving the way to several interesting ideas for future work. Yet, another potential area of exploration — from the point of view of model development — is to inform the model with source code-related features, because test plan quality may sometimes be dependent on the code changes that are being tested. Due to the limited dataset and the sparse nature of source code data, we leave this as future work.

Moreover, we have shown that RoBERTa-based approaches have the advantage of being more flexible and generalizable, while achieving better performance than a rule-based decision tree. However, despite the fact that the Matching Network architecture provides a lightweight notion of interpretability to some extent through the support set examples, the challenge of incorporating a fine-grain notion of model explanation remains as future work.

ACKNOWLEDGMENTS

The authors thank Katherine Zak, Dave Sukhdeo, Wes Dyer, and Drew Carlson for their helpful discussions that prompted this analysis and for their contributions to early iterations of this work.

REFERENCES

[1] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*. Ieee, 1–6.

[2] V.R. Basili, F. Shull, and F. Lanubile. 1999. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering* 25, 4 (1999), 456–473. https://doi.org/10.1109/32.799939

[3] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 146–156.

[4] Software Engineering Technical Committee et al. 1983. IEEE Standard for Software Test Documentation. *Institute of Electrical and Electronic Engineers Computer Society* (1983).

[5] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. 2015. Code reviews do not find bugs. how the current code review best practice slows us down. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 27–28.

[6] Nhan Cach Dang, María N Moreno-García, and Fernando De la Prieta. 2020. Sentiment analysis based on deep learning: A comparative study. *Electronics* 9, 3 (2020), 483.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *North American Association for Computational Linguistics (NAACL)* (2018).

[8] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian network classifiers. *Machine learning* 29, 2 (1997), 131–163.

[9] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–23.

[10] Nathan Hilliard, Lawrence Phillips, Scott Howland, Artëm Yankov, Courtney D Corley, and Nathan O Hodas. 2018. Few-shot learning with metric-agnostic conditional embeddings. *arXiv preprint arXiv:1802.04376* (2018).

[11] Touseef Iqbal and Shaima Qureshi. 2020. The survey: Text generation models in deep learning. *Journal of King Saud University-Computer and Information Sciences* (2020).

[12] Oleksii Kononenko, Olga Baysal, and Michael W Godfrey. 2016. Code review quality: How developers see it. In *Proceedings of the 38th international conference on software engineering*. 1028–1038.

[13] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. 2019. Text classification algorithms: A survey. *Information* 10, 4 (2019), 150.

[14] Andrey Krutauz, Tapajit Dey, Peter C Rigby, and Audris Mockus. 2020. Do code review measures explain the incidence of post-release defects? *Empirical Software Engineering* 25, 5 (2020), 3323–3356.

[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[16] Mika V Mäntylä and Casper Lassenius. 2008. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering* 35, 3 (2008), 430–448.

[17] Sameen Maruf, Fahimeh Saleh, and Gholamreza Haffari. 2021. A survey on document-level neural machine translation: Methods and evaluation. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[18] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 192–201.

[19] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 171–180.

[20] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. 2002. Thumbs Up? Sentiment Classification Using Machine Learning Techniques. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 79–86.

[21] Nicolas Papernot and Patrick McDaniel. 2018. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765* (2018).

[22] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[23] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[24] Yasir Ali Solangi, Zulfiqar Ali Solangi, Samreen Aarain, Amna Abro, Ghulam Ali Mallah, and Asadullah Shah. 2018. Review on Natural Language Processing (NLP) and its toolkits for opinion mining and sentiment analysis. In *2018 IEEE 5th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. IEEE, 1–4.

[25] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2020. How to Fine-Tune BERT for Text Classification? arXiv:1905.05583 [cs.CL]

[26] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2017. Review participation in modern code review. *Empirical Software Engineering* 22, 2 (2017), 768–817.

[27] Suresh Thummalapenta, Pranavadatta Devaki, Saurabh Sinha, Satish Chandra, Sivagami Gnanasundaram, Deepa D Nagaraj, Sampath Kumar, and Sathish Kumar.

2013. Efficient and change-resilient test automation: An industrial case study. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1002–1011.

[28] M. Onat Topal, Anil Bas, and Imke van Heerden. 2021. Exploring Transformers in Natural Language Generation: GPT, BERT, and XLNet. arXiv:2102.08036 [cs.CL]

[29] Thomas P Trappenberg. 2019. Machine learning with sklearn. In *Fundamentals of Machine Learning*. Oxford University Press, 38–65.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[31] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. 2016. Matching networks for one shot learning. *Advances in neural information processing systems* 29 (2016), 3630–3638.

[32] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537* (2019).

[33] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A multi-task benchmark and analysis platform for natural language understanding. *n International Conference on Learning Representations (ICLR)* (2018).

[34] Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 38–45.

[35] Yu Zhou, Michael Wursch, Emanuel Giger, Harald Gall, and Jian Lu. 2008. A bayesian network based approach for change coupling prediction. In *2008 15th Working Conference on Reverse Engineering*. IEEE, 27–36.