A System-Wide Debugging Assistant Powered by Natural Language Processing

Pradeep Dogga
UCLA

Karthik Narasimhan
Princeton University

Anirudh Sivaraman
NYU

Ravi Netravali
UCLA

ABSTRACT

Despite advances in debugging tools, systems debugging today remains largely manual. A developer typically follows an iterative and time-consuming process to move from a reported bug to a bug fix. This is because developers are still responsible for making sense of system-wide semantics, bridging together outputs and features from existing debugging tools, and extracting information from many diverse data sources (e.g., bug reports, source code, comments, documentation, and execution traces). We believe that the latest statistical natural language processing (NLP) techniques can help automatically analyze these data sources and significantly improve the systems debugging experience. We present early results to highlight the promise of NLP-powered debugging, and discuss systems and learning challenges that must be overcome to realize this vision.

KEYWORDS

systems debugging, natural language processing

ACM Reference Format:
Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, and Ravi Netravali. 2019. A System-Wide Debugging Assistant Powered by Natural Language Processing. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3357223.3362701

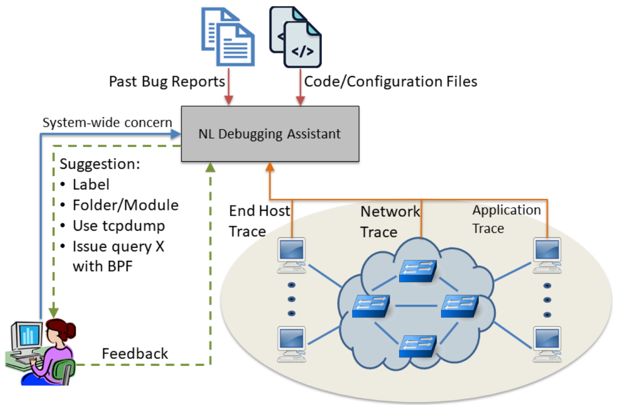

Figure 1: Overview of how our proposed debugging assistant improves different steps in a developer's end-to-end debugging workflow. Developers begin by submitting system-wide bugs or performance concerns to the NL debugging assistant. The assistant generates hints (e.g., files to investigate) or actions (e.g., debugging queries to issue) based on analyzing past bug report data, design documents, tracing information collected throughout the system, and developer input. The process is iteratively followed until a bug is resolved.

1 INTRODUCTION

Despite the proliferation of debugging techniques and tools [4, 21, 28, 29, 31, 38, 45, 49], diagnosing and fixing bugs in systems remains challenging. The debugging process for developers typically entails reading a bug report, understanding its context in the system, reproducing the bug, iteratively asking and answering questions to identify its root cause, and then developing and testing potential fixes. Much of this process is manual, has many false starts, and requires significant developer familiarity with the system at hand—an increasingly elusive requirement as systems get more complex.

Indeed, given the increasingly powerful systems debugging tools we now have, a key challenge for developers is identifying how to leverage the fine-grained operations they support (e.g., queries) when presented with a high-level bug or issue. For example, what debugging tool should the developer use when a service is deemed unreachable? What query should they issue?

At the same time, coupled with the increase in system and debugging complexity, there has been an important shift in the way that developers write and run systems, tackle bugs, and document their experiences. In particular, the amount of *auxiliary data* associated with software systems has rapidly grown, e.g., monitoring logs, execution traces, bug reports, patches, and code documentation. These data sources embed helpful debugging information [16], but existing debugging tools fail to fully leverage them, and instead place the burden of extracting insights from this data on developers.

There are two reasons why this auxiliary data isn't fully exploited in today's debugging workflows. First, the auxiliary data sources, in contrast to source code, are highly unstructured and varied, ranging from custom logging formats (e.g., from debugging tools) to natural language bug reports. Techniques for automatic test generation [11, 30] and program repair [24, 42] rely on precise definitions of correctness and cannot fully leverage such unstructured data.

Second, in large systems, problems occur at the intersection of different system components. The above approaches, as well as existing monitoring and debugging tools, only instrument specific subsystems (e.g., network switches [18, 37, 38], end-host network stacks and operating systems [4, 35, 45, 46], or applications [39, 49]), but still place the onus of correlating information across subsystems on developers. Thus, existing tools have limited utility in debugging issues that arise due to the interaction between subsystems. For example, they are unable to automatically determine that an application timeout was the result of a routing black hole, since they fail to link together data from multiple disparate sources (in this case, network and application-level traces from an entire cluster).

Our thesis is that natural language processing (NLP) techniques are well-positioned to analyze the large amount of auxiliary data across multiple subsystems and extract insights that significantly enhance the current debugging experience. The natural language nature of auxiliary data and the inexact nature of the debugging process require models that go beyond existing techniques that need exact and structured inputs. NLP can provide these models.

Prior work has demonstrated the promise of NLP in debugging scenarios (§2), but has focused on specific sub-problems (e.g., generating Bash scripts from natural language [25] or extracting keywords from bug reports [43]). Though promising, we believe there is a significant untapped opportunity for systems research that integrates these NLP techniques—and develops new ones—into a system-wide interactive debugging assistant that facilitates and accelerates each step in a developer's end-to-end debugging process (Figure 1). In §3, we identify three features that are crucial to such a system: a) preliminary diagnosis of incoming bug reports, b) automatic generation of debugging queries (for existing debugging tools) to monitor different subsystems, and c) taking multiple diagnostic or corrective decisions towards fixing the bug.

We performed several sets of preliminary experiments (§4). First, using a diverse set of 98 open-source GitHub repositories, we trained text classification models to automatically predict summaries of solutions (i.e., developer-provided labels for issues) and repository contents (i.e., folders relevant to the issue) from high-level repository issues reported in English. Our models predicted labels with 76% precision and 78% recall, and folders with 74% precision and 76% recall. Figure 2 shows concrete examples of the helpful suggestions that our models can provide. Second, we created a local distributed systems testbed and an associated fault injector running Reddit [3] over an emulated Mininet topology [23] with the Marple [38] debugging tool integrated in. We performed template-based generation by training a classifier to map from user-written issue text to structured Marple debugging queries; precision and recall were 82% and 67%, respectively. These early results show the promise of NLP techniques to learn and generalize from diverse data sources to produce accurate predictions for various stages of the debugging process.

In summary, we outline a vision for an NLP-based debugging assistant that will ease systems debugging by:

- automatically extracting debugging insights from unstructured auxiliary data (e.g., bug reports, code comments, execution traces, monitoring logs) to diagnose system-wide bugs.
- helping developers better leverage state-of-the-art debugging tools (e.g., network debuggers, distributed system tracers, and end-host monitors) by suggesting which tools to use, what low-level queries to issue to each one, and when to do so.

Collectively, these tasks will accelerate the process of *identifying the root cause* of a bug or issue. We envision our assistant becoming an integral part of the developer's ecosystem, as compilers, code editors, and interpreters are today.

2 RELATED WORK

Program analysis and synthesis: NLP techniques have been utilized in multiple aspects of software development [14], through the lens that software systems comprise not only source code statements, but also information-rich natural language comments that should not be ignored. Examples include detecting operations with incompatible variable types [19] and converting natural language comments into assertions [15]. More recently, NLP has also been used in code generation by allowing developers to specify requirements in high-level natural language. This entails parsing loosely organized input in natural language to generate structured output in the forms of regular expressions [26], Bash programs [25], API sequences [17], and queries in domain specific languages [12].

Program debugging: NetSieve [43] used NLP to parse network trouble tickets by generating a list of keywords and using a domain-specific ontology model to extract ticket summaries from those keywords. While NetSieve automates parsing, significant manual effort is still required in (1) offline construction of an ontology model and (2) determining what constitutes a keyword. In contrast, we seek to build models that learn automatically from data, with minimal manual effort. Net2Text [8] translates English queries into SQL queries, issues those queries, summarizes the results, and translates them back into natural language for easy interpretation. We aim to go further and automatically determine which queries to issue based on bug reports, debugging traces, and source code. Recent bug localization work uses information retrieval techniques [22, 36, 53], but requires manual feature engineering.

Big Code: Recent efforts such as the Big Code initiative [50] perform statistical program analysis to take advantage of the large amount of code in existence today, with the goal of extracting insights to aid code generation, refinement, and debugging. Learning techniques have been used to identify comments that are largely redundant with source code [27] or generate natural language summaries of source code [40, 52]. We view work in the Big Code initiative as contributing to some of the individual building blocks of our proposed debugging assistant. However, significant effort is required to integrate these building blocks.

Summary: The aforementioned projects offer a glimpse into the ability to extract meaning from natural-language auxiliary data present in software projects. They also illustrate the benefits of combining information from natural language and source code [14]. However, these tools fall short of realizing our vision. First, these approaches are limited to ingesting data from a single subsystem. However, our target distributed systems scenarios require extracting and relating diverse data (e.g., bug reports, source code, code comments, execution traces) from multiple subsystems (e.g., the network and applications on end hosts). Second, all of these approaches assume a single-step process, where the NLP system has to perform a single prediction. In contrast, we focus on the end-to-end system debugging process that is iterative by nature.

3 EXPEDITING DEBUGGING WITH NLP

The manual debugging workflow:

Debugging usually starts with a bug report filed by a bug reporter (e.g., network tickets [16], JIRA, or GitHub issues). This bug report is specified in natural language (e.g., English). The text of the report is analyzed by a developer to decide whether it is truly a bug, as opposed to a feature request or a configuration error at the bug reporter's end. If it is a bug, it is assigned to some developers, who then progressively narrow it down to a particular subsystem, folder, file, and eventually a change to some lines of code.

Viewed end-to-end, the debugging workflow takes as input the text of the bug report and outputs either a bug-fix patch representing changes to the source code, a comment telling the bug reporter how to fix their configuration error, a comment saying that the bug won't be fixed, or a comment saying that the feature request won't be handled. Our goal is to reduce the end-to-end latency of this workflow, i.e., from the time the bug is reported to the time it is deemed handled, by using NLP to automate some debugging steps.

Opportunities for automation: We now identify opportunities to augment the developer's debugging workflow with automation at different points during the debugging process. Developer intuition is invaluable to the debugging process and complementary to the automation afforded by techniques from NLP. Conceptually, imagine an NLP-powered debugging assistant running in the background to continuously ingest text from various sources: the bug report's text, the bug report's comments as they come in, the source code of the repository, and different traces. It then produces recommendations, e.g., assign a particular label to the bug report, look at this folder to diagnose the bug. These recommendations are shown to the developer, who can act on them and fix the bug, or provide further input to the debugging assistant based on their domain knowledge. This assistant can be applied to several different parts of the workflow:

1. *Preliminary diagnosis*: At the beginning of the workflow, the assistant can be used to perform diagnostics on incoming issues such as assigning labels or localizing the relevant subsystems in the project at various granularities, from top-level directories to individual lines of code. The assistant can also assign each ticket to the most relevant developer in order to streamline reviews. These tasks broadly fall into the framework of text classification and document retrieval. However, our setup presents a unique challenge of learning joint representations of the data that capture useful information from both unstructured text and structured source code.

2. *Generating debugging queries*: Another part of the workflow involves the assistant generating domain-specific queries to monitor different subsystems (e.g., BPF code for monitoring the kernel). This falls under the umbrella of language generation which plays a key role in problems like machine translation or text summarization. The added challenge here lies in learning a model that can effectively use a diverse set of sources like issue text, system status information, and source code semantics to generate useful and syntactically correct debugging queries (e.g., queries in BPF [6] or AppDynamics [5]).

3. *Active (interactive) debugging*: Finally, the debugging assistant could take multiple diagnostic or corrective actions, each building upon previous actions and their results. For this, we will draw upon techniques for sequential decision making like reinforcement learning, with the goal being to perform the optimal sequence of actions to fix the problem, within constraints on the latency of performing these actions or the compute resources expended in the process. In addition, we can keep the developer in the loop to supervise the entire process, and simultaneously help fine-tune the assistant's decision making. We imagine that developer and assistant actions will be interleaved to debug the issue efficiently and with minimal human effort.

Ultimately, our vision is an assistant that captures the hard-won debugging wisdom of the expert programmer in different parts of the workflow by exploiting the abundant data available within source code repositories and their associated issue trackers. Of course, across all of these use cases, we must provide system support to ensure a developer-assistant interface that seamlessly handles expressive input options from the developer and provides timely responses from the assistant. In particular, developers must be able to (but not required to) input case-specific debugging information (e.g., time limits, expertise levels), and receive responses in a way that does not add undue latency to any debugging stage.

4 EXPERIMENTS

We present early evaluations for three automation techniques that we developed to expedite different stages of the end-to-end debugging workflow: predicting labels for a GitHub issue, predicting folders in a GitHub repository that are relevant to an issue, and generating debugging queries in a real distributed system.

4.1 Label and Folder Prediction

Label prediction for project issues: We formulate the label prediction task as a standard text classification problem. Formally, consider a dataset $\mathcal{D} = (x, y)$ containing pairs of issue text (x) and their corresponding labels (y). Since each issue may have more than one valid label, our task is a multi-label classification problem [47]. Therefore, we consider each y to be a one-hot vector of size $|L|$, where L is the set of all possible labels, with each entry in the vector being 1 or 0 depending on whether a particular label applies to the issue or not. Our goal is to train a model to accurately predict as many labels as possible.

The key challenge lies in learning an appropriate representation for the bug reports available in textual form. This representation should be able to capture the semantics of the issue sufficiently to predict accurate labels. There are several techniques and models that have shown considerable promise in NLP such as word embeddings [32, 41] or LSTM recurrent neural networks [20], which convert discrete textual symbols into a real-valued vector representation. As an initial foray, we use a bag-of-words representation to convert the text in each issue into a suitable vector: $\phi(x)$. Each entry in this vector corresponds to the number of times a particular word appears in the text (most entries will be 0). We train a linear classifier f to predict probabilities for each label from this representation:

$$f_\theta(\phi(x)) = W \cdot \phi(x) + b; \qquad \hat{y} = \sigma(f_\theta(\phi(x)))$$

where W is a matrix of weights, b is a bias vector, σ is the ReLU function and $\hat{y}$ is a vector of predictions over all labels. We train our model by minimizing the binary cross-entropy loss (over all the labels) with respect to θ using stochastic gradient descent [10]:

$$\mathcal{L}(\theta) = -\sum_i [y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$$

Prediction of source code folders: Moving down a level of granularity, we also consider the task of predicting files/folders that might

Label prediction	# issues	165966 (15)
	mean # words/issue	137 (117)
	# distinct words in dataset	98786 (497)
Folder prediction	# issues	240138 (25670)
	# distinct words in dataset	15225 (4011)
	# folders	1706 (174)

Table 1: Statistics on our dataset of GitHub issues. Each entry is the value summed across all repos and for the median repo (in parentheses).

be relevant to a particular issue, and would be useful for a developer to look at. This is a challenging task to automate since it requires a semantic understanding of both natural language text as well as the semantics of each folder (and its contents, e.g., source code).

As a first step, we focus on predicting folders that appear in changelists linked to issues. This is inherently similar to the problem of information retrieval, where the goal is to return relevant documents given a natural language query. In our case, the the issue is a query and our goal is to return a (ranked) list of relevant folders in the project. Our key requirement is to learn good representations and a similarity metric between issues and folders,

Assume each instance in our dataset is a pair (x,z), where each x is an issue text and each z is the text corresponding to a single folder (e.g. folder name). Our goal is to learn a similarity function $\psi(\phi_1(x), \phi_2(z); \theta)$ which can be used to predict relevant folders given a new issue x'. Here, the ϕs are again suitable representations for converting text into a real-valued vector; we use the same bag of words (BOW) representation as previously described. For the similarity function ψ, we train a 2-layer neural network that operates on the concatenation of both BOW vectors $[\phi_1(x); \phi_2(z)]$ to predict the probability of a folder being relevant: $\hat{P}(y=1|x,z)$.

We treat the pairs in the dataset as positive examples of matches, and generate pairs of negative examples ($\mathcal{D}' = (x', z')$), by randomly matching issues to a folder in the code that is not relevant. With this, we can train our model by minimizing the following loss function with respect to θ using stochastic gradient descent.

$$\mathcal{L}(\theta) = -\Big[\sum_{(x,z)\in D} \log \hat{P}(y=1|x,z) + \sum_{(x',z')\in D'} \log \hat{P}(y=0|x',z')\Big]$$

Setup: We perform empirical evaluation of our models on real-world data sourced from publicly available code repositories, specifically 98 repositories hosted on GitHub.

We collect text from closed issues along with associated labels and pull requests containing details on modified folders in the source code. This gives us supervised data for both classification problems.

For the label prediction task, we select the top-5 labels across all repositories according to their frequency of occurrence. Upon filtering for issues that contain these labels, we end up with 165,966 issues in total. We split this into train (72%), validation (8%) and test (20%) sets. We use the validation set to determine thresholds for classifying various classes by using basin-hopping [51]. For folder prediction, we take each issue and treat the corresponding folders referenced in the commit data as positive examples while randomly generating other folders from the repository as negative examples. Table 1 provides more details on our data; as shown, the problem is challenging due to the diverse vocabulary and large scale of data.

Results: We use standard classification metrics of precision, recall and F-1 scores. For the label prediction task, we also report accuracy, considering a case to be correct if the set of predicted labels exactly

- **Issue 1**: *I want to be able to access a specific resource variable within that resource. For example : run a provisioner for an instance and supply it with the instance private ip (or id or anything else).*
  True Labels: CORE, ENHANCEMENT
  Predicted Labels: CORE, ENHANCEMENT
- **Issue 2**: *The menu panel not being closed when its 'overlayref' is detached externally using 'detach' for example when using the 'closescrollstrategy'. **note:** this is a re-submit of #8654 due to some sync issues.*
  True Folder: src/lib/menu **Model score**: 0.99
  False Folder: src/tools/dashboard **Model score**: 0.0

Figure 2: Examples of label & folder predictions for two repos: hashicorp/terraform and angular/material2.

matches the set of true labels. For folder prediction, we also report scores for mean average precision (MAP), which is an aggregate metric over precision at various levels of recall. We achieve 77.8% accuracy and 0.77 F-1 on label prediction (Table 2), which is quite promising for a *multi-label* classification problem over 5 classes. For comparison, a random baseline would achieve 20% on a simpler *single-label* classification problem. On folder prediction, we achieve an F-1 score of 0.75 and a MAP score of 0.72 for predicting relevant folders from more than 160 folders in the entire repository. This is significantly higher than a random baseline which would get a MAP score of 0.0062. Figure 2 lists qualitative examples of our model predictions. These results highlight the ability of NLP techniques to relate varied data sources (i.e., by learning a model across repos that are quite different in terms of topic, code, and structure).

4.2 Query Generation

A second capability we target for our assistant is the ability to automatically generate syntactically correct queries for systems and network debugging tools (e.g., Marple [38], GDB [4]) to aid the human debugging process. We formulate this as a contextual language generation problem, where the system takes user-written issue text x as input[1] and generates a structured debugging query q. As an initial foray, we perform template-based generation [44], where we train a classifier (f) to predict the most relevant template $\mathcal{T}$ for an issue and then predict values for the slots in the template to generate q:

$$\hat{\mathcal{T}} = f(x); \hat{q} = g(\hat{\mathcal{T}}, x)$$

The classifiers are trained using ground truth data collected using our setup (described below), using cross entropy as the loss function during training. We stress that template-based generation is a first step. In the future, we plan to investigate more sophisticated generation models such as recurrent neural networks [20] and transformers [48], which have shown considerable promise in tasks like language modeling and machine translation.

Setup: We created a local testbed that runs the open source Reddit web application [3] over an emulated Mininet topology [23, 33]. Each of Reddit's service components, including Memcached [13], Cassandra [1], and PostgreSQL [2], was hosted on a separate container within the Mininet environment, and components were connected using emulated network switches that could be programmed in P4 [9]. Our setup incorporated the Marple network debugging

[1]The input could also consist of other signals like system status.

- **Issue**: *Took a while and the webpage says 'You broke reddit' and 'Funny 500 page message 3'. Upon refreshing the page it says 'Funny 500 page messsage 6'. Upon further loads, the browser is stuck on 'waiting for 10.0.0.2'*
- **Actual Fault**: mcrouter instance down.
  Relevant Query: `stream = filter(T, switch==1); result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);` **Model Score**: 0.94
  Irrelevant Query: `stream = filter(T, switch==5); result = groupby(stream, [srcip, dstip, srcport, dstport, proto], count);` **Model Score**: 0.01

Figure 3: **Example query predictions for debugging a given issue.**

framework [38], which assigns each network switch and packet a unique ID, and supports queries that (1) determine per-packet and per-switch queueing delays, and (2) aggregates various metrics (including user-defined ones) obtained across packets. Marple queries are compiled by the Marple compiler into P4 programs that can be run on the emulated switches inside Mininet.

We then designed an automatic fault injector, which randomly injects faults into our Mininet topology. Faults include inducing component failures, creating congestion on various links, and adding NAT and firewall router rules that create routing loops or drop certain traffic. To produce realistic debugging data, we configured the fault injector to inject a single unknown fault into our setup. We then (without knowledge of which fault was selected) acted as end users and developers. As end users, we recorded observations of different Reddit web pages (e.g., page load delays, missing content) to create the user-written issue text. As developers, we iteratively issued Marple queries until the root-cause was identified. In total, we collected 28 sets of end-user issues, Marple queries, and query outputs; we split this into train (72%), validation (8%) and test (20%) sets. Our classifier considers two kinds of query templates: one which counts packets for a given 5-tuple identifier, and another which counts per-packet queueing delay and switch queue length. Both templates are parameterized by the switch ID.

Results: Our results show early promise for template-based query generation. We achieve a MAP score of 0.82 and an F-1 score of 0.76 for generating relevant Marple queries (Table 2). Figure 3 shows an example query output for a scenario where a system component has failed. As shown, the model correctly predicts both the appropriate query template to use and the switch to issue it on.

Extending to other tools: Though our evaluation focused on the Marple debugging tool, we believe that the techniques used naturally generalize. We observe that most debugging frameworks ultimately support SQL-like frameworks for querying, e.g., the Jaeger [7] distributed tracing system supports SQL-like querying of its traces. By extracting this unified structure from the languages/grammars of different tools, we can represent any query generation problem as a sequence generation problem. This extensibility is similar to how compilers are re-engineered to target different backends.

5 CHALLENGES AND FUTURE WORK

Systems challenges: All of the above use cases require a developer-assistant interface that seamlessly handles expressive input options

	Precision	**Recall**	**F-1**
Label Prediction	0.76	0.78	0.77
Folder Prediction	0.74	0.76	0.75
Query Generation	0.82	0.67	0.76

Table 2: **Results for label and folder prediction, as well as template-based query generation tasks.**

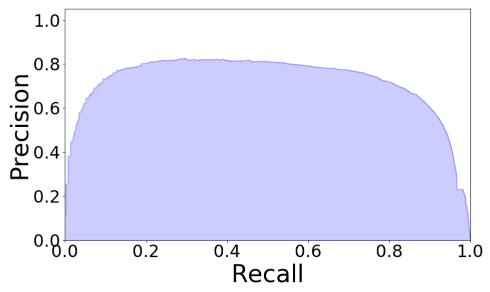

Figure 4: **Precision vs. recall for folder prediction.**

from the developer and provides timely responses from the assistant. Keeping the developer in the loop is critical because the developer represents years of domain expertise that is complementary to the data-driven NLP approach. However, determining the right interface is challenging as developer time and inputs are scarce resources. Repeatedly asking the developer for inputs negates automation benefits, but judicious developer inputs (e.g., a hint that the bug might reside in a particular subsystem) could significantly improve the assistant's output. It is also important to determine what developer inputs to request. For instance, in a multi-label classification problem, the developer could provide a rank-ordered list of labels, which can be converted to a prior probability distribution over the labels. Further, since developer time is precious, the developer should be able to smoothly tradeoff prediction accuracy or granularity for lower prediction time, e.g., predicting issue labels is likely faster than predicting the files associated with an issue.

Machine learning challenges: While our preliminary experiments use a bag-of-words representation, determining the optimal model to use requires additional work. For instance, we can capture more of the compositionality in text with models like RNNs, CNNs, or even the recently proposed Transformer model [48]. However, such advanced models have high training overheads with respect to input data size, processing time, and memory. For widespread use, novel systems and algorithmic techniques must be developed to perform such training in resource-constrained settings. Potential approaches include distributed training, incremental real-time inference, and model predictions conditioned on developer preferences. There are also challenges related to data availability and privacy as systems can span multiple organizations; supporting such scenarios would require algorithmic techniques like privacy-preserving machine learning [34].

End-to-end evaluation: Our evaluation measured standard classification metrics because this was a preliminary investigation. However, the real value of such an assistant will be a reduction in end-to-end debugging latency and a commensurate improvement in the developer experience. To understand the impact on end-to-end debugging latency, we plan to conduct a developer study using the debugging assistant on systems of reasonable complexity (e.g., TensorFlow).

REFERENCES

[1] 2016. Apache Cassandra. http://cassandra.apache.org/.

[2] 2016. PostgreSQL. https://www.postgresql.org/.

[3] 2016. reddit. https://github.com/reddit/reddit.

[4] 2018. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[5] 2019. AppDynamics Query Language - ALY310 | University | AppDynamics. https://learn.appdynamics.com/courses/appdynamics-query-language-aly310.

[6] 2019. Berkeley Packet Filter - Wikipedia. https://en.wikipedia.org/wiki/Berkeley_Packet_Filter.

[7] 2019. Jaeger: Open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[8] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. 2018. Net2Text: Query-Guided Summarization of Network Forwarding Behaviors. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 609–623.

[9] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM CCR* (July 2014).

[10] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 177–186.

[11] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[12] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM.

[13] Dormando. 2015. Memcached-a distributed memory object caching system. https://memcached.org/.

[14] Michael D. Ernst. 2017. Natural Language is a Programming Language: Applying Natural Language Processing to Software Development. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*. 4:1–4:14.

[15] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM.

[16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM)*. ACM.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.

[18] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[19] Irfan Ul Haq, Juan Caballero, and Michael D. Ernst. 2015. Ayudante: Identifying Undesired Variable Interactions. In *Proceedings of the 13th International Workshop on Dynamic Analysis (WODA 2015)*. ACM.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Jonathan Kaldor, Jonathan Mace, Michal Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*. ACM.

[22] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.

[23] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*. ACM.

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 38, 1 (Jan. 2012), 54–72.

[25] Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 2018. NL2Bash: A Corpus and Semantic Parser for Natural Language Interface to the Linux Operating System. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC)*.

[26] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation of Regular Expressions from Natural Language with Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

[27] Annie Louis, Santanu Kumar Dash, Earl T. Barr, and Charles A. Sutton. 2018. Deep Learning to Detect Redundant Method Comments. *CoRR* abs/1806.04616 (2018). arXiv:1806.04616 http://arxiv.org/abs/1806.04616

[28] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)*. ACM.

[29] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. ACM.

[30] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2018. Predictive Test Selection. *CoRR*

abs/1810.05286 (2018). arXiv:1810.05286 http://arxiv.org/abs/1810.05286

[31] James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*.

[32] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[33] Mininet Team. 2018. Mininet An Instant Virtual Network on your Laptop (or other PC).

[34] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE, 19–38.

[35] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and Precise Triggers in Data Centers. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM)*. ACM.

[36] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 382–391.

[37] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[38] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[39] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[40] Jayavardhan Peddamail, Zhen Wang, Ziyu Yao, and Huan Sun. 2018. A Comprehensive Study of StaQC for Deep Code Summarization. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Lond, UK, August 2018*.

[41] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[42] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*. ACM.

[43] Rahul Potharaju, Navendu Jain, and Cristina Nita-Rotaru. 2013. Juggling the Jigsaw: Towards Automated Problem Inference from Network Trouble Tickets. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[44] Ehud Reiter and Robert Dale. 2000. *Building natural language generation systems*. Cambridge university press.

[45] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2015. CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM.

[46] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with Pathdump. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. USENIX Association.

[47] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining (IJDWM)* 3, 3 (2007), 1–13.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[49] Nicolas Viennot, Siddharth Nair, and Jason Nieh. 2013. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of ASPLOS*.

[50] William W. Cohen, Charles Sutton, and Martin T. Vechev. 2016. Programming with "Big Code" (Dagstuhl Seminar 15472). (01 2016). https://doi.org/10.4230/DagRep.5.11.90

[51] David J. Wales and Jonathan P. K. Doye. 1997. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A* (1997).

[52] Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. 2018. StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*.

[53] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.