AutoTSG: Learning and Synthesis for Incident Troubleshooting

Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula,
Arjun Radhakrishna, Anurag Gupta
{t-mamola,chetanb,saupa,arradha,anugup}@microsoft.com
Microsoft

ABSTRACT

Incident management is a key aspect of operating large-scale cloud services. To aid with faster and efficient resolution of incidents, engineering teams document frequent troubleshooting steps in the form of Troubleshooting Guides (TSGs), to be used by on-call engineers (OCEs). However, TSGs are siloed, unstructured, and often incomplete, requiring developers to manually understand and execute necessary steps. This results in a plethora of issues such as on-call fatigue, reduced productivity, and human errors. In this work, we conduct a large-scale empirical study of over 4K+ TSGs mapped to 1000s of incidents and find that TSGs are widely used and help significantly reduce mitigation efforts. We then analyze feedback on TSGs provided by 400+ OCEs and propose a taxonomy of issues that highlights significant gaps in TSG quality. To alleviate these gaps, we investigate the automation of TSGs and propose `AutoTSG` – a novel framework for automation of TSGs to executable workflows by combining machine learning and program synthesis. Our evaluation of `AutoTSG` on 50 TSGs shows the effectiveness in both identifying TSG statements (accuracy 0.89) and parsing them for execution (precision 0.94 and recall 0.91). Lastly, we survey ten Microsoft engineers and show the importance of TSG automation and the usefulness of `AutoTSG`.

1 INTRODUCTION

At Microsoft, we operate services at a massive scale with 1000+ internal and external services built and operated by tens of thousands of engineers spread across the world and deployed in over 200 data centers worldwide. At such a large scale, we need effective incident management processes to minimize the impact of service incidents. Today, most software companies have on-call duty, which requires engineers building services to be responsible for handling (i.e., acknowledge, diagnose and mitigate) incidents 24x7 on a rotating basis. To standardize these incident management workflows, engineering teams document these steps as *Troubleshooting Guides* (TSGs) which are then referred to and followed by the on-call engineers while handling production incidents. These TSGs help with knowledge sharing and avoid the challenges with tribal knowledge, especially when new engineers join the team.

In Microsoft, we have more than 50,000 TSGs which are used regularly for incident resolution by over 60,000 engineers every month. The TSGs are authored by the engineers and can contain various components such as commands and scripts for troubleshooting, big data queries for fetching diagnostics logs, natural language instructions and even screenshots. At the time of incident handling, the on-call engineer manually tries to follow the instructions described in the TSGs. The manual execution can consume a significant amount of effort since commands/queries must be copy-pasted and executed, and instructions must be parsed and understood. Further, similar to other kinds of software documentation [1], TSGs

are also prone to various issues such as lack of readability, fragmentation, etc. These issues have significant detrimental impact on both engineering productivity and service health because it leads to increased effort by the on-call engineers and, also, higher customer impact due to increased incident resolution time. Further, with manual TSGs, there is a significant risk of outages[1] due to human errors, on-call fatigue, and knowledge gaps. Hence, there is a need to automate the TSGs into executable workflows, which can help mitigate incidents with minimal human intervention.

In this work, we conducted a large-scale empirical study to understand the usage and challenges of TSGs better. We find that incidents linked with TSGs have reduced mitigation time showing the effectiveness of TSGs. At the same time, TSGs are also prone to completeness, validation and maintenance issues. To mitigate these problems and reduce the manual effort and human error involved in executing TSGs, we investigate the problem of automating TSGs by converting them to executable workflows such as Jupyter notebooks. We propose **AutoTSG**, a novel framework for TSG automation at scale. Executable TSGs help minimize the manual effort for the on-call engineer while also improving the maintainability with automated testing and validation. With `AutoTSG`, our goal is to assist developers with automation of 50,000+ TSGs at Microsoft. To overcome the unique challenges in this task, such as lack of labelled data and heterogeneity of information embedded in TSGs, we combine meta-learning and program synthesis for extraction of components (i.e., code, big data queries, natural language instructions, etc.) and parsing of components into constituents such as variables, commands in code and conditional, action statements in natural language. Our evaluation shows that `AutoTSG` has high accuracy while also being useful based on the survey of on-call engineers. To summarize, we make the following main contributions in this work:

(1) We do a large-scale empirical study on the usage and effectiveness of TSGs for incident resolution at Microsoft.
(2) We analyze feedback provided by 400+ on-call engineers at Microsoft to propose the first taxonomy of TSG quality issues.
(3) We design and build `AutoTSG`, a novel framework which combines machine learning and program synthesis to aid with the automation of TSGs at scale.
(4) We do a quantitative evaluation on 50 TSGs and survey 10 Microsoft engineers to show the effectiveness of `AutoTSG`.

The rest of the paper is organized as follows: In Section 2, we present insights from the empirical study about the usage of TSGs and motivate the need for automation. In Section 3, we provide an overview and the implementation details for `AutoTSG`. In Section 4,

---

[1] https://www.wsj.com/articles/amazon-finds-the-cause-of-its-aws-outage-a-typo-1488490506

Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula,
Arjun Radhakrishna, Anurag Gupta

(a) Dist. of #incidents per TSG

(b) #TSGs vs Incident Severity

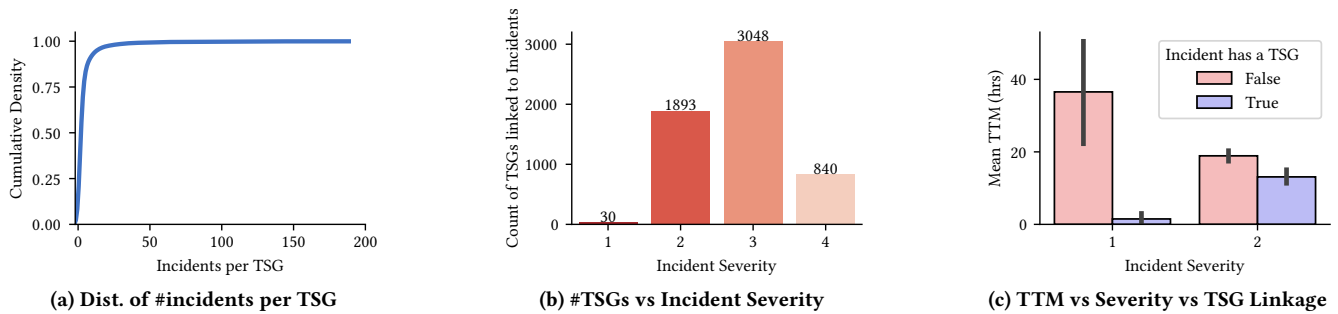(c) TTM vs Severity vs TSG Linkage

Figure 1: Analysis of TSG Usage

we describe the experimental evaluation and user study for `AutoTSG`. In Section 5, we discuss the related work followed by conclusion.

2 EMPIRICAL STUDY

In this section, we empirically study two aspects of TSGs – *usage* and *quality*. First, we characterize how TSGs are used for incident mitigation, using factors like usage frequency, incident severity, and time-to-mitigate (TTM). We analyze a large dataset of incidents mapped to TSGs by actual click-throughs on links to the TSG. We find new insights indicating that TSGs are widely used, and that incidents linked to TSGs have significantly lower mitigation time.

Then, we perform a large-scale study of 400+ feedback items provided by 100s of developers on TSGs at Microsoft. Our findings indicate significant gaps in quality aspects like *Completeness*, *Correctness*, and *Up-to-dateness*. Our results also uncover new and granular issues, such as *Broken Links* and *Empty* TSGs. Lastly, we discuss the implications of our findings and recommend actions to tackle these quality challenges. Further, we propose a new direction – *TSG Automation* – that guides our vision to improve the state of incident troubleshooting.

2.1 TSG Usage

We first collect a large dataset of incidents that have TSGs linked to them. Here, we use click-through data on links to TSGs to map incidents to corresponding TSGs. Unlike prior work [24] that study recommended TSGs and their effect on reducing effort, this approach captures the actual usage of TSGs by on-call engineers and strengthens our findings. Consequently, our dataset of incidents to TSG mapping is not static but rather represents actual click-throughs performed on-call for incident mitigation. Our dataset contains over 1000s of incidents[2] mapped to ≈**4800** TSGs, collected over a 4 month period. We then analyze this dataset from various perspectives to answer the **RQ**: *How are TSGs used for incident management?*

Incidents per TSG. To understand how frequently TSGs are used for mitigating incidents, we analyzed the distribution of the number of incidents per TSG in our dataset. As shown in Figure 1a, in 4 months, ≈47%, 17%, and 8% of the TSGs had at least 2, 5, and 10 incidents linked to them, respectively. We also observed six TSGs

linked to 100+ incidents each, the highest being 184. These results show that TSGs are frequently used to mitigate recurrent incidents.

TSGs and Incident Severity. In Figure 1b, we look at the total number of TSGs linked to incidents of each severity level. At Microsoft, incidents are classified into 4 severity levels: (1) Sev 1: Outage, (2) Sev 2: High, (3) Sev 3: Medium, (4) Sev 4: Low. Here, Sev 1 and 2 are *paging incidents*; i.e., an on-call engineer is alerted as soon as the incident occurs. Here, we find a small number of TSGs linked to outages (30). This is expected considering they are less frequent and may require deeper analysis and mitigation strategies. Next, we find that 85% of TSGs are linked to either high (1893) or medium (3048) severity incidents. Lastly, we find a relatively lower 14% of TSGs linked to low severity (840) incidents that are non-urgent and have no SLA impact. This shows that TSGs are commonly linked to critical incidents that affect multiple services and impact SLAs.

TSGs and Time-to-mitigate. Lastly, we analyze the effect of TSGs on efficient incident mitigation. We study the relationship between the time-to-mitigate (TTM) of incidents and whether the incident had a TSG linked (TSG Linkage). Here, we make sure to analyze incidents of Severity 1 and 2 only, since on-call engineers are notified immediately after their occurrence, and hence, TTM, is a reliable proxy for on-call effort. We find that the mean TTM of incidents without TSGs (≈19 hrs) is significantly higher than that of incidents linked to a TSG (≈13 hrs). Further, in Figure 1c, we analyze the correlation of this finding with incident severity. Here, we observe that severity 1 incidents linked with TSGs (≈36 hrs) had considerably lesser TTM than those without TSGs (≈2 hrs). For severity 2, we see a similar pattern where the mean TTM is reduced from ≈18hrs to ≈13hrs on linking TSGs to incidents. Overall, our analysis indicates that TSGs tend to significantly reduce effort during incident mitigation.

2.2 TSG Quality

In Section 2.1, we show that TSGs are key to incident mitigation. However, in an internal survey of on-call experience at Microsoft, developers picked *TSG Quality & Coverage* as the top pain-point out of 19 dimensions studied including volume of alerts, timing, and tooling. Based on developer feedback, we find that TSGs are prone to issues such as missing information, incorrect steps, and

---

[2]We cannot disclose the number of incidents due to Microsoft Policy.

Table 1: Taxonomy of intents for feedback on TSGs and their frequencies

Feedback Intent	Description	Examples	Frequency
Completeness	TSG is missing information like examples, links, etc.	'unknown impact and mitigation', 'please provide examples'	32.24%
Broken Link	TSG has broken or invalid links.	'use cases links are broken.', 'link leads to "404 - not found"'	13.32%
Correctness	TSG has incorrect or misleading information.	'information is wrong', 'steps didn't work'	11.21%
Readability	Feedback on TSG clarity, conciseness, grammar, etc.	'too much info, not organized', 'confusing terminology'	10.28%
User Exp. (UX)	TSG accessibility, navigation, formatting, etc.	'how to execute those code cells?', 'badly formatted query'	10.05%
Empty	TSG is empty or has dummy content.	'this page is empty', 'fill in the TSG, currently just has TODO'	7.24%
Up-to-dateness	TSG content is outdated.	'out of date: still using visualstudio instead of ado', 'deprecated'	6.54%
Relevance	Whether TSG is relevant to the user's issue.	'doesn't tell me how to renew my cert', 'nothing useful here'	3.97%
Other	Unclear intent or outside the scope of TSG quality.	'nice page', 'loved this page'	5.14%

Table 2: Taxonomies of documentation quality in prior work

Study	Taxonomy
Aghajani *et al.* [1, 2]	Correctness, Completeness, Up-to-dateness, Maintainability, Readability, Usability, Usefulness, Doc. process, Doc. tools
Plösch *et al.* [54]	Accuracy, Clarity, Consistency, Readability, Structuredness, Understandability, Completeness, Conciseness, Concreteness, Modifiabiality, Objectivity, Writing Style, Retrievability, Task Orientation, Traceability, Visual Effectiveness
Garousi *et al.* [19]	Completeness, Organization, Including visual models, Relevance, Preciseness, Readability, Accuracy, Consistency, Up-to-date, Examples

outdated content. In this section, we thus aim to empirically answer the **RQ**: *How do developers perceive the quality of TSGs?*

Setup. In this study, we analyze the quality aspects of TSGs through feedback provided by developers and on-call engineers at Microsoft. To this end, we first collect a dataset of feedback provided on TSGs over 4 months. Each feedback item contains (1) Thumbs-up/down rating and (2) a free text message to help improve the TSG. Our dataset contains **428** feedback items (👍:61, 👎:367), for **394** unique TSGs. Using this, we first develop a taxonomy for the feedback intent using the open coding approach. We then map each feedback item to an intent category and analyze their distributions.

We start by splitting our dataset into 3 sets: (1) **87** items from month 1, (2) **119** items from month 2, and (3) **222** items from months 3 & 4. Then, the first two authors of this work used the open coding approach to label the first set. They assigned a label to each feedback item based on what they perceived as the most prominent intent behind it. Subsequently, they discussed the feedback categories and agreed on a common taxonomy. Next, they independently labeled the second set to make sure no new categories emerged. They then had another discussion to settle disagreements and define a common understanding of each category.

Lastly, they annotated the third set and computed the inter-annotator agreement score using Cohen's kappa [12]. The resulting Cohen's kappa score was 0.907 indicating near-perfect agreement. Here, disagreements were mostly because multiple categories applied to certain feedback. Such disagreements were resolved by picking (1) the intent that occurs first in the feedback and (2) the most specific intent. With this approach, the annotators settled all disagreements and created a conclusively labeled dataset, which was used for all further analyses. In Table 1, we show the resulting taxonomy with descriptions of the intent and examples.

Prior Work. While we focus on understanding TSG quality, there has been prior work investigating different aspects of software documentation. Particularly close to our work, are empirical studies on generic software documentation quality that either (1) survey software practitioners or (2) mine stackoverflow and github data, to manually create a taxonomy of issues. In comparison, we perform a large-scale study of 400+ feedback items provided by 100s of developers, explicitly collected to improve the TSGs. Further, we collect feedback directly where an on-call engineer would visit to use the TSG for mitigation. As a result, we observe unfiltered feedback, enabling an accurate study of developer issues.

Table 2 shows some taxonomies developed in prior empirical studies [1, 2, 19, 54] of software documentation quality. Here, we observe some overlap with all prior work, but we also introduce new categories and rename a few others. Particularly, we share significant overlap with the taxonomy defined by Aghajani et al. [1, 2]. We retain 4 of their 9 categories – Correctness, Completeness, Up-to-dateness, and Readability. Further, we rename Usability to User Experience and Usefulness to Relevance, making them more appropriate to the experience of using TSGs to mitigate incidents. We also add specific categories essential to the quality of TSGs – Empty and Broken Link. Lastly, we do not include the Documentation process and tools categories, as our study focuses on TSG content only.

Feedback Distribution. Table 1 shows the frequency of feedback categories in our dataset. As shown, *Completeness* (32.24%) is the most frequent, and includes issues pointing to missing information such as steps, examples, links, and points of contact. The second most frequent category is *Broken Link* (13.32%). Here, feedback indicated errors while accessing links such as 404-not found, 403-forbidden, page not found, etc. Thirdly, we have *Correctness* (11.21%), where the feedback indicated misleading/incorrect information such as conflicting steps, incorrect steps, erroneous queries/commands, etc. We observe that these top-3 categories speak to the actual content in the TSG, accounting for a noteworthy total of 56.77%.

Next, we have *Readability* (10.28%) and *UX* (10.05%), that account for ≈20% of the feedback. This shows that a significant portion of quality issues faced by users is associated with how TSGs are presented and used, not just their content. Following that, we have *Empty* (7.24%) and have *Up-to-dateness* (6.54%) issues that reveal important maintainability issues with the current state of TSGs.

Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula,
Arjun Radhakrishna, Anurag Gupta

Table 3: Types of Components in TSGs

Component: Description	**Example**
ADF [44]: Link to an Azure Data Factory, a data integration service.	http://adf.azure.com/factory=resourceGroup/y/.../factories/z
Jarvis [45]: Link to a Jarvis dashboard, an internal telemetry platform.	https://jarvis.msft.net/dashboard/share/xxx
Kusto [46]: Database query in Kusto Query Language (KQL).	StormEvents | where State == "FLORIDA" | count
Powershell [48]: Statements from CLI scripts built on .NET Runtime.	$tenant = "<your tenant id/name>"
Torus: Secure Powershell scripts to manage Azure resources and datacenters.	$rules = Get-TransportRule -Organization $org
Merlin: Custom Powershell scripts to diagnose and fix Sharepoint issues.	Update-GridTenantProvisioningStamp $TenantId
Natural Language: Other natural language instructions.	*If the status is green, the problem is self-resolved.*

Lastly, we find feedback pointing to TSGs lacking *Relevance* (3.97%) to the user's needs. This is a critical issue for troubleshooting, as developers follow the steps in TSGs to mitigate incidents.
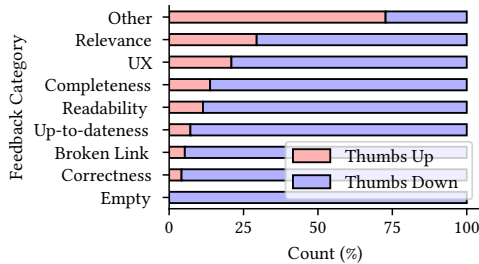
Figure 2: Distribution of Ratings for each Category

Rating Distribution. Each feedback item also contained a thumbs-up (positive) or thumbs-down (negative) rating for the associated TSG. Overall, we find that a majority 367 (85.7%) items had thumbs-down, and 61 (14.3%) had thumbs-up ratings. Prior work has shown that this is expected as people tend to give negative feedback far more than positive [66]. In Figure 2, we look at the distribution of these ratings within each feedback category. As shown, all major categories (i.e., leaving the *Other* category) have a majority of negative ratings associated with feedback on TSGs. Further, we looked at the negativity ratio for each category, i.e., the ratio of 👎 to 👍 votes.

We find that feedback for *Empty* were (naturally) always associated with negative feedback, and hence had the highest negativity ratio. Following that, we have *Correctness* (23:1), *Broken Link* (18:1), and *Up-to-dateness* (13:1) with very high negativity. Next, we have *Readability* (8:1) and *Completeness* (6:1) with high attributed negativity. Then, with relatively lower ratios are *UX* (4:1) and *Relevance* (2:1). Lastly, the *Other* category had the lowest negativity ratio of 0.38:1; i.e., more positive feedback than negative, as shown by examples in Table 1. Overall, our results indicate that on-call engineers and developers negatively perceive multiple aspects of TSG quality.

2.3 Implications & Recommendations

Our TSG usage analysis shows that TSGs are frequently linked to recurring incidents: 14% TSGs with >5, and six TSGs with 100+ incidents in four months. Secondly, we find that TSGs are most commonly (>85%) linked to critical severity incidents and significantly

reduce mitigation time and effort (avg: 6hrs). Our findings indicate that TSGs are essential and widely used documentation that aid the mitigation of incidents, especially the ones which frequently repeat.

However, our results on the quality of TSGs highlight important issues. We find that the top-3 most frequent feedback categories were *Completeness*, *Broken Link*, and *Correctness*, that account for 56% of all feedback. TSG users have to deal with missing steps, lack of explanations, incomplete examples, invalid links, incorrect commands, etc. Next, we find that presentation and UX also strongly affect quality; i.e., 20% of feedback points to *Readability* or *UX* issues stemming from verbose language, poor formatting, and organization. In context, next, we provide recommendations to alleviate these issues from our industry experiences (indicated by ⚡).

⚡ **Maintenance & Testing:** A common solution to mitigate the Completeness, Broken Link, and Correctness issues can be to introduce maintenance and testing for TSGs. This is challenging considering TSGs are mostly siloed text documents (Word, OneNote, etc.) and the executables in them need to be parsed. Hence, software research should invest in automated tools that can help parse, review, test, and maintain software documentation, similar to source code.

⚡ **Centralization & Standardization:** To solve issues with Readability and User Experience, we make 2 recommendations that are being enforced at Microsoft: (1) Converting all TSGs to a standard format (e.g., Markdown) with clear formatting guidelines. (2) Adopting a unifying platform for organizing TSGs, with an accompanying search engine. We find that these recommendations are consistent with those provided by prior work like Aghajani et al. [2].

⚡ **User-in-the-loop Review:** Additionally, assuming an enforced TSG review pipeline, we believe that involving users of TSGs in the review process is essential. This can solve readability issues where TSG authors assume clarity, but the users find it difficult to read and use the TSG. From our experience at Microsoft, teams can adopt this since most TSG users are fellow developers in the team who take up on-call rotation at a specified frequency (e.g., once a month).

How do we get there? While we provide some insights and recommendations to improve TSG quality, we note significant research challenges. For instance, as previously stated, it is non-trivial to apply reviewing, testing, and maintenance techniques to semi-structured and informal text documentation like TSGs. However, we observe that the software engineering domain has effectively dealt with improving quality issues in source code. That brings us

to the question: *Can we automate TSGs and bring them closer to source code?* We observe that this change can in-turn introduce the recommendations we make into the world of TSGs, such as code review, regression testing, and version control. With this, we envision TSGs of the future to be verified semi-automated (like jupyter notebooks) or fully-automated workflows, that can be executed with minimal manual touches. Such automated TSGs would reduce manual toil, minimize human errors, and also improve DRI health.

3 `AutoTSG`: TOWARDS TSG AUTOMATION

Towards this vision of automated TSGs, in this section, we introduce `AutoTSG`, a tool to aid the automation of manual TSGs to executable workflows. Particularly, `AutoTSG`'s design is guided by three unique observations about TSGs:

(1) **TSGs contain components** that are distinct pieces of information. As shown in Table 3, they are commands, database queries for logs, dashboard links, instructions, etc.
(2) **TSGs have control flow** resembling a decision tree. For e.g., if conditions mentioned in natural language.
(3) **Components contain constituents** that are parts of a component expected to be parsed for execution. For e.g., command name and parameters for a Powershell command.

These observations guide us to design a two-phase framework for automating troubleshooting guides that first identifies TSG components (Component Identification) and then parses them to extract constituents necessary for execution (Component Parsing). There has been significant research on text/code identification and parsing in prior work [32, 65]. However, applying them directly to TSGs requires addressing some unique challenges:

(1) How do we identify components? While heuristic-based methods lack coverage, most supervised learning methods need 1000s of labeled examples to train accurate classifiers. In the context of TSGs, this means manually labeling 1000s of examples for 10s of component types. This is laborious and limits the scalability of our tool to new component types. Hence, we need appropriate models that can learn from a limited set of labeled examples.

(2) How do we parse components? One approach is to hand-craft parsers to extract each constituent of a component. But it assumes significant domain expertise and manual effort. On the other hand, ML based parsers require large training datasets and are stochastic. Hence, we need to also automatically learn verifiable parsing programs from a small set of examples.

In the rest of the section, we describe the design of the `AutoTSG` framework, as shown in Fig. 3, that address these challenges.

3.1 Component Identification

Overview. Toward TSG automation, we need to first extract and identify individual statements in TSGs that need to be automated – *Components* – such as commands, database queries, dashboard links, and also natural language instructions. Here, for automation, it is not only important to extract these statements but also to identify the component type. Table 3 shows examples of some popular components used in TSGs at Microsoft.

Intuitively, we can view component identification as a supervised classification problem – given a statement, classify it into a category.

However, most supervised learning techniques require thousands of labeled examples for training. In the context of TSGs, this implies manually labeling 1000s of examples for 10s of component types. Further, this poses challenges to the scalability of `AutoTSG` to new components types with very few examples. Hence, we solve this using a few-shot learning setup, where we aim to learn models with a minimal amount of training examples.

But, there are challenges with learning models from very few examples, such as overfitting, robustness, and generalizability. We alleviate this by moving away from the classical learning framework (learning to classify) to a meta-learning framework, where we *learn how to learn* to classify. First, we train a Siamese neural network on a meta-task – learning how similar or different components are. Then, we use a nearest-neighbor search approach to identify component types. In the following subsections, we describe our approach to component identification in detail.

3.1.1 **Preprocessing**. Today TSGs are decentralized and can be in various formats such as Word, Markdown, OneNote, etc. Therefore, we first use the `pandoc` library [40] to convert all TSGs to a single format that is parsable programmatically – Markdown. Next, we clean the converted TSG using regexes by pruning information that is non-trivial to parse, such as images and tables. Here, we plan to extend `AutoTSG` to capture such multi-modal information with tools such as optical character recognition (OCR). Then, we segment the TSG into statements using newline characters. Here, we use some simple heuristics to handle multi-line commands and queries. For instance, we remove indentation around the '|' character for `Kusto` queries and around the '{' and '}' (braces) for command-line scripts. Lastly, we tokenize TSG statements into tokens. Here, we use a custom implemented tokenizer to handle camel-case, URLs, command names, and multi-line database queries.

3.1.2 **Meta-Learning Framework**. As previously stated, in this setting, we need to learn a component classifier from a minimal set of examples, i.e., a few-shot learning setup. To enable few-shot learning, we use meta-learning [58] as a framework to simplify the component identification task to a meta-task. Meta-learning [58], commonly understood as *learning to learn*, refers to an outer (meta) algorithm updating an inner learning algorithm such that the model it learns improves an outer objective (meta-task).

Here, we choose our meta-task as *"Given a pair of statements, predict the probability that they belong to the same component type"*. The intuition here is that one way to learn classification is to differentiate between component types. For instance, to learn to classify Powershell commands and SQL queries, we can learn what makes Powershell and SQL similar or dissimilar. Then, based on learnt properties, we can decide which component type is the closest match.

In meta-learning literature, this translates to metric-based meta-learning [28]. The core idea in metric-based meta-learning is similar to nearest neighbors algorithms (e.g., k-NN classifier [4] and k-means clustering [37]), where the predicted probability P over a set of labels y is a weighted sum of labels of support set samples S. The weight is generated by a kernel function k_θ, measuring the similarity between two data samples.
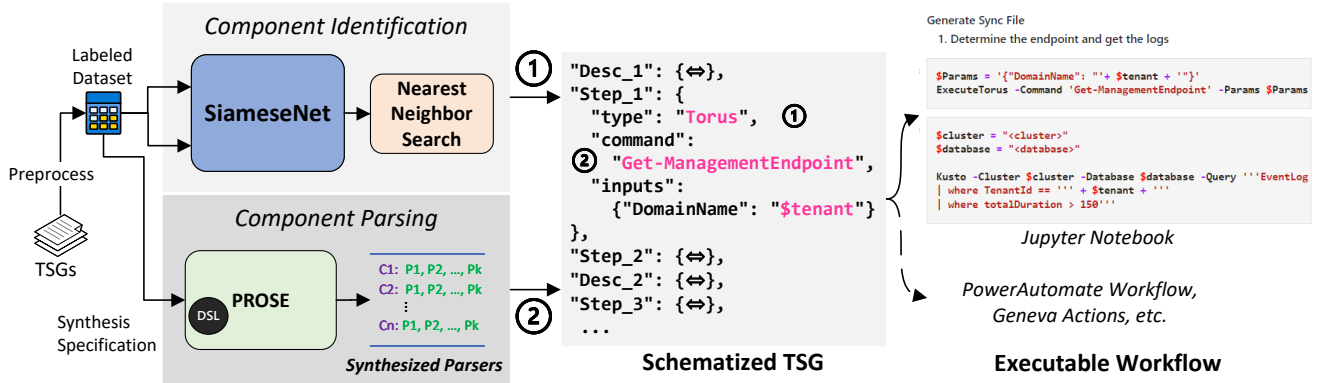
Figure 3: Overview of AutoTSG pipeline

$$P_\theta(\boldsymbol{y}|\boldsymbol{x}, S) = \sum_{(x_i, y_i) \in S} k_\theta(x, x_i) y_i \quad (1)$$

Our aim here is to learn a kernel k_θ that is aligned with learning a similarity metric over component types. Hence, we learn to map TSG components to a latent (embedding) space, where different components are well separated and similar components are close.

3.1.3 ***Siamese Network****.* To learn our meta-learning task, we use a Siamese Network (SiameseNet) architecture proposed by Koch et al. [28]. The Siamese Network architecture contains two twin networks and a distance metric, that are jointly trained to learn the relationship between pairs of input data samples. The twin networks are identical and share the same weights and parameters. The SiameseNet accepts two inputs $\mathbf{x}_a$ and $\mathbf{x}_b$, which are featurized inputs of the same or different component types. For featurization, we use the simple yet effective bag-of-words [22] approach which creates one-hot features. Next, a convolutional neural network [33] learns to encode the 2 resultant vectors via an embedding function f_θ. Here, f_θ contains a 100 dim. embedding layer, 2 convolutional layers [33], 2 max pooling operations [7], and a 128 dim. dense layer.

The L_1 distance between the two resultant embeddings is $||f_\theta(\mathbf{x}_a) - f_\theta(\mathbf{x}_b)||_1$. But, to decide whether the two inputs are drawn from the same component type, we need to convert this unbounded distance to a probability p. We do that by computing the exponent of the negative L1 norm (equation 2). Finally, as shown in equation 3, we train our SiameseNet on a binary cross-entropy loss function as the network label $\mathbf{y}$ is binary. Here $\mathbf{y} = 1$ whenever $\mathbf{x}_a$ and $\mathbf{x}_b$ are of the same component type and $\mathbf{y} = 0$ otherwise.

$$p(\mathbf{x}_a, \mathbf{x}_b) = exp(-||f_\theta(x_a) - f_\theta(x_b)||_1) \quad (2)$$

$$\mathcal{L} = \sum_{\forall(\mathbf{x}_a, \mathbf{x}_b, \mathbf{y})} \mathbf{y} \log p(\mathbf{x}_a, \mathbf{x}_b) + (1 - \mathbf{y}) \log(1 - p(\mathbf{x}_a, \mathbf{x}_b)) \quad (3)$$

3.1.4 ***Nearest neighbour Search****.* With a trained SiameseNet based meta-learner, we can now identify the probability of a pair of statements belonging to the same component. In other words,

we can now use the SiameseNet as a comparator that returns the similarity between a pair of TSG statements. With this, a naive approach to performing component identification would be to compare a given TSG statement against every training example. Then, we can label the TSG statement with the component type of the most similar training example – the nearest neighbor.

But, performing this comparison for each TSG statement with every training example is quite inefficient (O(#training-examples × #statements)), especially for longer TSGs. To mitigate this, for each component type $c \in C$ we pre-compute a *prototype* $\mathbf{prot}_c$ as the mean of the embedded training examples S_c of that component type:

$$\mathbf{prot}_c = \frac{1}{|S_c|} \sum_{(\mathbf{x}_i, y_i) \in S_c} f_\theta(\mathbf{x}_i) \quad (4)$$

Now, for a TSG statement we only compare against each prototype and identify the label as the component type associated with the nearest neighboring prototype. This improves the efficiency of our nearest-neighbor search (O(#component-types × #statements)).

3.2 Component Parsing

From the techniques of Section 3.1, we obtain a list of components in a TSG and their respective component types. In this section, we learn *component parsers* for each component type to extract the constituents of the component. For example, given a Powershell command, the component parser will extract the command name and the parameters.

The main challenge here is handling the wide variety of components that commonly occur in TSGs. Further, large companies and organizations often use components written in custom scripting languages that are unique to only the organization. Hence, the diversity in component types and the presence of custom component languages make hand-crafting a generic set of parsers infeasible.

Instead of hand-crafting a set of component parsers, AutoTSG uses program synthesis techniques to learn parsers from examples. Specifically, we use programming-by-example techniques to learn parsers from a small number of user-provided examples. Here, for each component type, the user provides example components along with their expected constituents, and the program synthesis engine

produces a parser to extract these constituents in the form of a python program. Programming-by-example techniques are specifically suitable in this scenario as they are able to learn from few (between $1 - 5$) examples, and can produce efficient, deterministic, and user-readable parsing programs. In comparison to machine learning techniques, this both avoids the need for a large amount of training data and further allows the user to edit the produced parser program to fix any minor issues.

Programming-by-Example in AutoTSG. We do not describe the full program synthesis procedure and instead provide a high-level overview of the PROSE program synthesis library and how AutoTSG uses PROSE. The reader is referred to [21] for a literature survey on program synthesis. PROSE implements an extension of the Flash-Fill [20] programming-by-example technique. Given a set of examples of the form $in_i \mapsto out_i$ where in_i and out_i are strings, PROSE produces a program P (and its translation to Python) such that $P(in_i) = out_i$ for all i. The programs produced by PROSE are of two kinds: (a) single-branch expressions that are concatenations of constant strings and substrings of the input string, and (b) conditional expressions over single-branch expressions. The conditions in the conditional expressions and the substrings for single-branch expressions are built-up using standard string and regular expression operators such as StartsWith, EndsWith, IndexOf, Regex.Find, etc. The reader is referred to the documentation of PROSE for the exact class of programs that can be synthesized [47].

Example 3.1 (Single-branch programs). Consider the Powershell assignment statements below.

```
1  $mb = Get-Mailbox senderOrRecipientMailbox
2  $tenant = "<your tenant id/name>"
3  EOP: $rulePackage = Get-DlpSensitiveInformation -Org ...
```

Note that the assignment statement (3) has a label associated with it ("EOP"). For this component type, one constituent of interest is the left-hand side of the assignment, i.e., the variable being assigned to. To synthesize the parser for this constituent, the user provides 3 examples of the form $(1) \mapsto$ "$mb", $(2) \mapsto$ "$tenant", and $(3) \mapsto$ "$rulePackage", where (i) represents the corresponding assignment statement from above. Given these examples, AutoTSG uses the program synthesis library PROSE [47] to synthesize the following python program.

```
1  def prog0(self, s):
2    idx1 = s.index("$")
3    idx2 = re.search(r"[$][\p{L}0-9]+", s).end()
4    return s[idx1:idx2]
```

This program slices the input assignment statement between the first occurrence of a dollar symbol up to the end of the first sequence of alphanumeric characters preceded by a dollar symbol.

While the above example can be handled using simple regular expressions, the synthesizer is able to produce more complex parsers that involve conditionals and other sophisticated operations.

Example 3.2. Consider the list of Kusto query components below.

```
1  TbaFilteringException | where time > ago(1d) | ...
2  cluster('Aznwautotriage').database('autotriage').
      AutoTriageIcmNer | sort by IncidentId desc
```

From these components, we are interested in extracting the constituent representing the table name, i.e., "TbaFilteringException" and "AutoTriageIcmNer", respectively. Given the examples $(1) \mapsto$ "TbaFilteringException" and $(2) \mapsto$ "AutoTriageIcmNew", PROSE generates a conditional program with two branches, each handling one example.

```
1  def prog0(self, s):
2    if re.match("^cluster", s):
3      idx1 = s.rindex(".") + 1
4      idx2 = re.search(r"\p{Zs}+", _0).start()
5    else:
6      idx1 = re.search(r"[-.\p{L}0-9]+", _0).start()
7      idx2 = re.search(r"\p{Zs}+", _0).start()
8    return s[idx1:idx2]
```

This program handles both formats of Kusto query components gracefully. In practice, there are several more variations of the Kusto query component that occur in TSGs and the program generated by PROSE has more branches to deal with them–we present this two branch version here for simplicity.

Table 4 presents some input and output pairs for different components and corresponding constituents, along with the description of the parsing program produced by PROSE.

Special component types. While most component types and constituents can be handled using the above techniques, we discuss 2 special cases which require additional procedures.

Handling natural language. Traditional program synthesis techniques are not designed to handle the complexities of natural language. For example, consider the component *"If you need to force the file sync, you can use ForceSync parameter"*. From this component, we are interested in extracting the *condition clause* constituent ("you need to force the file sync") and the *action clause* constituent ("can use ForceSync parameter"). Using PROSE directly will force us to rely on fragile punctuation based parsers such as "extract between the word If and the first comma" which would then fail on differently punctuated statements like *"If it is due to any other error contact the reporting team"*.

To avoid learning such fragile rules, we first annotate the clauses in the input component using a constituency tagger (see, for example, [27, 53]) to annotate 3 kinds of constructs: simple declarative clauses, subordinate clauses, and verb phrases. The tagged version of the component is *"If <CL1>you need to force the file sync</CL1>, <CL2>you can use ForceSync parameter</CL2>"*. With this tagged component, the program synthesizer is able to synthesize a simple parser that relies on searching for the anchor points <CL1>, </CL1>, <CL2>, and </CL2>. This parser would also handle components with missing punctuation correctly as the constituency tagger natively understands natural language and does not rely on punctuation.

Iterative constituent extractions. In certain component types, some constituents appear repeatedly. For example, in a Powershell command, parameter name and value constituents occur as many times as there are parameters in the command. In the command Test-PolicyDistributionStatus -Org nybc.com -PolicyId 8dbdfce9 -Verbose True, there are 3 parameter names ("$-$Org", "$-$PolicyId", and "Verbose") and 3 parameter values ("nybc.com", "8dbdfce9", and "True").

Manish Shetty, Chetan Bansal, Sai Pramod Upadhyayula,
Arjun Radhakrishna, Anurag Gupta

Table 4: Description of some component parsers synthesized from example

Component	Constituent	E.g. input	E.g. output	Description of synthesized parsing program
PowerShell	1st Param	`Test-PolicyDistributionStatus -Org nybc ...`	`-Org`	Extract between first two whitespace spans
ADF	Subscription	`https://.../subsc/SUB1/resourceGroups/...`	`SUB1`	Extract alphanumeric span preceding `/resourceGroups/`
		`Tba	where ...`	`Tba`	
Kusto	TableName	`cluster(...).db(...).AutoTriage	sort ...`	`AutoTriage`	Condition program with 3 branches
		`let result = newUser	where ...`	`newUser`	
NL Conditional	Condition	`If it returns True, you can ...`	`it ... True`	Extract between tags `<CL1>` & `</CL1>` after constituency tagging

One potential way of handling this scenario is to learn a different parser for the ith parameter for each i. While sound, this strategy would repeat work learning a similar parser for each i. Instead, we follow an iterative strategy in `AutoTSG`: we only learn parsers for the 1st constituents. During extraction, we (a) first extract the constituents for the first parameter from the component obtaining " $-$ Org" and "nybc.com", (b) delete the extracted constituents from the component to obtain the new component `Test-PolicyDistributionStatus -PolicyId 8dbdfce9 -Verbose True`, and (c) repeat the steps as long as there are more constituents to be extracted obtaining in sequence " $-$ PolicyId", "8dbdfce9" and " $-$ Verbose", "True".

4 EVALUATION

4.1 Component Identification Evaluation

Setup. To evaluate `AutoTSG`'s component identification model, we use a manually labeled dataset. We begin by first extracting **1902** statements from **50** TSGs from various services at Microsoft. We then manually classify these sentences into their respective component types, using a combination of domain expertise and existing command databases. For evaluation, we choose **7** component types (shown in Table 3), based on their frequency of occurrence in TSGs as reported by domain experts at Microsoft.

To mitigate the explosion of data and class imbalance, caused by a large number of natural language instructions in TSGs and our pair-wise sampling approach during meta-learning, we limit `Natural Language` to **200** random examples. With this, we create a dataset of **661** labeled examples. We then compare `AutoTSG`'s few-shot SiameseNet model (Section 3.1.3) against multiple baselines, in a 5-fold cross-validation setting that ensures models do not over-fit training data. In Table 5, we report the 5-fold cross-validation precision, recall, and F1 scores, for each component type. We also report overall aggregated metrics, including the accuracy of our models.

Baselines. First we have `KNN_BoW` – a K-nearest-neighbor [4] model using a Bag-of-words [22] as features. Next, with `RF_BoW`, we introduce a Random Forest [8] model, while keeping Bag-of-words as features. Here, we specifically choose these two models as they are simpler, yet learn classification similar to our `SiameseNet` – by separating classes from each other. Next, we have `KNN_W2V` and `RF_W2V`, where we retain the models, but update the Bag-of-words feature space to Word2Vec [49]. For Word2Vec models, we fine-tuned pre-trained models to a corpus of 3000+ TSG sentences, using `sentencepiece` [30] and `gensim` [55].

Baseline Results. From Table 5, we see that all baselines perform quite well overall. `RF_BoW` is the best, with an overall F1 of 0.81 and an accuracy of 0.78. However, we see that these models perform well for certain components, but poorly for others. For instance, in Table 5, we observe high F1 scores (0.71–0.93) for `ADF`, `Jarvis`, and `Kusto`. Table 3 shows that these components have distinct structure and syntax (e.g., 'https://adf.', 'https://jarvis-', '| where'), making them easier to identify. However, we find poor F1 scores (0.40–0.63) for components that are harder to distinguish, such as `Torus`, `Merlin`, and `Powershell`. This can be attributed to these components sharing syntax/structure, but having variations in vocabulary/semantics (refer Table 3). This makes distinguishing these components non-trivial and our baselines fail to capture these semantic variations.

SiameseNet. Lastly, we evaluate our proposed `SiameseNet` approach, which incorporates meta-learning to first learn a meta-task – distinguishing between components. We then utilize a nearest neighbor search approach to classify sentences into components in the embedding space. As shown in Table 5, our approach achieves an average accuracy of 0.89, which is significantly better than our baselines. We also observe that it reaches high F1 scores (0.72–0.98) across all component types. Particularly, unlike the baselines, we see strong results for components that are harder to distinguish like `Torus`, `Merlin`, and `Powershell`. Hence, with `AutoTSG`'s meta-learning approach capturing syntax and semantics of components, we are able to outperform multiple strong baselines.

4.2 Component Parsing Evaluation

Setup. Next, we evaluate `AutoTSG`'s effectiveness to parse and extract constituents of components using synthesized programs from PROSE. As described in Section 3.2, `AutoTSG` uses programming-by-examples to synthesizes parsers, for each constituent of a component (e.g., `parameters` of a `Powershell` command).

We then test these synthesized programs on the **1902** sentence labeled dataset described in Section 4.1 and collect the parsed outputs. Here, we ensure that there is no overlap between the specification examples and the test dataset. Then, we manually validate the precision (i.e., constituents parsed are correct) and recall (i.e., all constituents of the input are parsed) of each parsed output.

Results. Table 6 shows the average precision and recall of parsing, aggregated for each component. As shown, `AutoTSG`'s parser synthesizer can generate accurate parsers that can be learned from 5-10 specification examples. For instance, with just 8 examples, we can learn parsers for `Powershell` statements with a precision of 0.9

Table 5: Evaluation of Component Identification

Component	KNN_BoW			RF_BoW			KNN_W2V			RF_W2V			SiameseNet			Support
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	
ADF	0.65	1.00	0.75	0.90	1.00	**0.93**	1.00	1.00	1.00	0.87	1.00	0.90	0.67	1.00	0.72	6
Jarvis	1.00	1.00	1.00	1.00	1.00	1.00	0.69	1.00	0.78	0.82	0.77	0.75	1.00	1.00	**1.00**	14
Kusto	1.00	0.60	0.72	0.87	0.87	0.85	0.62	0.87	0.72	0.92	0.53	0.62	0.97	0.87	**0.91**	29
Merlin	0.60	0.51	0.54	0.69	0.49	0.55	0.45	0.42	0.43	0.49	0.38	0.43	0.95	0.65	**0.76**	106
Torus	0.70	0.84	0.76	0.75	0.84	0.79	0.54	0.78	0.64	0.52	0.72	0.60	0.82	0.95	**0.87**	202
Powershell	0.48	0.57	0.51	0.64	0.72	0.67	0.88	0.29	0.41	0.54	0.38	0.42	0.97	0.82	**0.87**	104
Natural Language	0.94	0.69	0.79	0.95	0.90	0.92	0.95	0.82	0.88	0.83	0.80	0.81	1.00	0.97	**0.99**	200
Overall	0.77	0.74	0.72	0.83	0.83	0.82	0.73	0.74	0.69	0.71	0.65	0.65	**0.91**	**0.90**	0.87	
Accuracy		0.69			0.78			0.62			0.65			**0.89**			

Table 6: Evaluation of Component Parsing

Component {Constituents}	Sup.	Pre.	Rec.
Torus {variable, command, parameters}	202	0.93	0.81
Merlin {variable, command, parameters}	106	0.92	0.86
Powershell {variable, command, parameters}	104	0.90	0.80
Kusto {cluster, database, table, query}	29	1.00	1.00
ADF {subscription, resourcegroup, factory}	6	1.00	1.00
Conditionals {condition, action}	127	0.87	0.97
Overall		**0.94**	**0.91**

and recall of 0.8. Also, with just 10 examples of conditional statements, we can learn parsers to extract conditions (   ) and actions (   ) accurately. More importantly, these parsers have high recall; i.e., are robust to variations in conditional statements such as *"If command returns True, then create an incident"*, *"If the status is False delete the resource"*, *"If average latency is > 300 ms"*, etc. Overall, we find that our parsers have a high average precision of 0.94 and recall of 0.91.

Further, we looked at some common test examples that were incorrectly extracted. For instance Torus parsers incorrectly parsed the statement: `$m = Get-Mailbox -Arbitrate -Identity $ identity` and returned {`variable`: `'$m'`, `command`: `'Get-Mailbox'`, `parameters`: `[('Arbitrate', '-Identity')]`}. As shown, the parsers incorrectly identified `Arbitrate` as a parameter whose value is `-Identity`. This is due to the usage of flag parameters like `-Arbitrate` in between commands, that were unseen in the synthesis specification. We find that these kinds of errors can be fixed by learning programs from a larger set of specification examples that cover these variations. In another example of a Powershell statement: `$m | Format-List $db`, we observed that the parsers returned empty results. This is because of the usage of pipe: `|` in a conjunction of statements, which was unseen in the specification. We find that these kinds of errors can be fixed with some preprocessing, such as splitting the command on pipes and iteratively calling the parsers.

Thus, our analysis shows that precise component parsers for TSGs can be effectively learned from a small set of examples through program synthesis – creating a scalable solution to expand AutoTSG to other kinds of components with minimal manual effort.
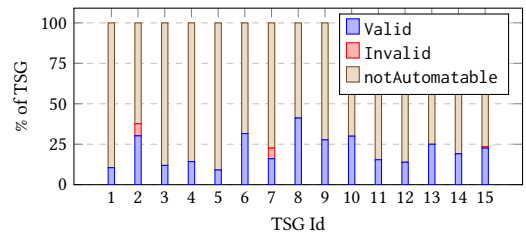
Figure 4: Coverage for TSG Automation

4.3 TSG Coverage Evaluation

Setup. In this section, we look at the overall coverage of AutoTSG for TSG automation. For this, we select the top **15** most frequently used TSGs for incident mitigation. We then run AutoTSG on these TSGs and validate the returned results for each line in a TSG – both the component type and parsed output. First, we mark all lines in the TSG that are notAutomatable; i.e., lines that cannot be converted to an executable. Next, for every automatable line, we verify if the result is Valid or Invalid for automation; i.e., whether the line was correctly identified and parsed by AutoTSG. Finally, we report the coverage of AutoTSG as the percentage of valid automatable lines.

Results. Figure 4 shows the results of this evaluation. First, we observe that TSGs have majority of notAutomatable lines. This is expected as much of a manual TSG is made up of statements that are either not executable or not required for automation. For example, in TSGs #1, #3, #4, and #5, we observe statements such as section headers, dates, links to other TSGs, author name, step descriptions, comments, and points of contact. While we accurately identify these statements as Natural Language, we cannot parse them into executables, hence, making them notAutomatable.

Next, we find that on average 21.24% of all lines in TSG is correctly automatable (Valid) using AutoTSG. Also, for 6 TSGs, more than 25% of all lines are correctly automatable, with a maximum coverage of 41.2% for TSG#8. Overall, we observe that for 12/15 TSGs, 100% of automatable lines in a TSG were correctly identified and parsed by AutoTSG showing that AutoTSG is highly effective at accurately identifying and parsing necessary information to automate a TSG into executable workflows.

4.4 AutoTSG User Study

So far, we have shown that AutoTSG has very high accuracy and coverage. Next, we perform a user study to understand the importance of TSG automation and the usefulness of AutoTSG.

Setup. In order to select the study participants, randomly sampled 30 on-call engineers at Microsoft who have mitigated or resolved incidents in the last six months and invited them for interviews. With a response rate of 33%, 10 out of the 30 invitees agreed for an interview. These 10 participants (P_1-P_{10}) work across 8 teams at Microsoft. They had mitigated or resolved on average 1852 incidents, with the minimum being 41 and maximum 5432. We conducted ≈15-minute semi-structured interviews with these 10 participants.

We began by asking how often the participant uses TSGs for incident mitigation. Next, we recorded verbatim responses on the issues they face with today's TSGs. Then, we introduced a manual TSG and asked if automating the TSG would help. If the response was yes, we introduced the participant to the results of TSG automation using AutoTSG and a human-in-the-loop approach. First, we showed a schematized TSG (JSON), returned by AutoTSG, with components identified and parsed – as in Fig. 3. Then, we showed the final automated TSG (Jupyter notebook) that would be programmatically generated from the schematized TSG. Given this, we asked the participants to rate the usefulness of AutoTSG for TSG automation on a 5-point Likert scale [56]. However, if the participant responded that automating the TSG would not help, we collected responses on why they thought so. Next, we summarize the results of the study.

Q1. How often do you use TSGs for mitigation? Here, a majority of six participants said they use TSGs every other week, while three others said they use them every month. Interestingly, one participant said they use TSGs every day. When asked why they use TSGs every day, they said:

P_2: *"My usual on-call rotation is bi-weekly. But I end up helping my peers use TSGs every day, because they are bad and confusing."*

Q2. What challenges do you face with TSGs? Here, we observed that all participants responded with some type of challenge they faced due to quality issues in TSGs. We find the majority of them talked about completeness, readability, and usability issues. Here are some representative verbatim responses:

P_4: *"TSGs maybe maintained, but we don't find them elaborative. SME (subject matter expert) thinks this is well defined, but for us or new members additional info should be added."*

P_6: *"I work on a service with a lot of customization. So when incidents arise, I have to manually verify which steps in a TSG will work for that incident. There is no guaranteed solution."*

P_7: *"Mostly outdated TSGs or missing the information I need. So I generally discuss with my team for many of the issues."*

Q3. Do you think automating TSGs would help? Here, notably, all participants of the user study said that automating TSGs would help. Apart from reducing on-call load, effort, and human error, the participants identified other positive outcomes of automation:

P_2: *"The team being able to review automated TSGs is very valuable. We can check for the safety of steps – which today is missing."*

Q3.a. If Yes, how useful is AutoTSG for TSG Automation? Here, we look at the distribution of the 5-point Likert scale ratings provided by the participants. We find that the participants gave the usefulness of AutoTSG an average rating of 4.2. A majority six participants gave a rating of 4, three participants gave a rating of 5, and one participant gave a neutral rating of 3. Overall, we find that our participants strongly perceived AutoTSG to positively aid the automation of TSGs. Some even remarked at how AutoTSG motivates an automated on-call experience of the future:

P_1: *"This tool also motivates teams to better document their queries, commands, etc., from the get-go in executable notebooks."*

P_3: *"When will this be available? This is great! This can drive future data analysis like which TSGs, commands, etc. were run frequently, and help find major issues."*

Q3.b. If No, why is automation not useful? As stated, no participant responded that automation of TSGs would not be useful.

5 RELATED WORK

Incident Management. Troubleshooting guides are critical for incident management, which has been a popular research direction in software engineering. Recent work has focused on multiple aspects of incident management like triaging [9, 10], mitigation [24], diagnosis [6, 39, 51], and more. Particularly close to our work, are efforts that attempt to mine structured knowledge from various artifacts, such as incident reports [26, 59, 60] and root cause documentation [57]. However, we tackle an aspect of incident management, that has received relatively lesser attention – TSGs. Jiang et al. [24], analyzed the usage of troubleshooting guides and proposed a TSG recommendation system to help developers find relevant TSGs. Our empirical study also supports the findings of such prior work. However, different from them, we also study the quality aspects of TSGs, like completeness, correctness, etc., that make them difficult to use. Lastly, our findings motivate the automation of TSGs, that in-turn introduces properties of source code to TSGs. We introduce AutoTSG – a novel framework to aid with the automation of TSGs.

Software Documentation. While studies on troubleshooting guides are limited, there have been several efforts to study software documentation in general. These can be classified into 2 categories: (1) tools to generate/recommend documentation and (2) empirical investigation of documentation usage and quality. Regarding automation for documentation, research has focused on either summarization or recommendation for bug reports [38, 41], code [13, 25, 42], user stories [29], API usage examples [23, 35, 50, 62], etc. Different from these, our tool AutoTSG focuses on automation that helps translate manual text documentation to executable workflows. Closer to our work in this space are the empirical studies on documentation. These studies use user surveys to analyze the importance and quality of software documentation [1, 2, 11, 14, 19, 54], but focus on software maintenance in general, unlike our work on the specific task of troubleshooting. Most analogous to our work on TSG quality is the taxonomy of documentation quality proposed by Aghajani et al. [2], to which we compare our work in detail in Section 2.2.

Few-shot Learning & Meta-Learning. Few-Shot Learning (FSL) [16, 17] is a type of machine learning problem, where we learn a

task from only a limited number of examples for the target. FSL is particularly useful to help reduce the burden of collecting large datasets of supervised information, such as in large-scale image classification [28, 64, 69], language modeling [68], drug discovery [3], robotics [15, 18], and more. Unlike these well studied scenarios, in this work, we use FSL in the domain of TSGs, with varying kinds of information – commands, queries, links, instructions, etc.

To enable FSL, we use meta-learning [58], commonly known as *learning to learn*. It refers to learning related tasks, and using this to learn new tasks much faster than otherwise possible [67]. Particularly, in our scenario, we use a flavour of meta-learning called metric-learning. Here, the idea is to learn input representations and a similarity metric during the meta-learning phase [28, 61, 63, 68]. In this work, we use a Siamese convolutional network [28] to embed TSG statements and separate the final task, component identification, from the neural net, and instead use a fast nearest neighbor search approach for classification, like Snell et al. [61].

Program Synthesis. Program synthesis techniques, especially programming-by-examples (PBE), have been applied to various domains [31, 34, 43]. Gulwani [20] introduced FlashFill to synthesize string transformation scripts from examples. In the software engineering domain, there have been efforts to apply program synthesis techniques on code related tasks such as learning version update patches [5], code edit scripts [43], and merge conflict resolutions [52]. However, our work targets a new domain that has not been explored using PBE: software documentation. Different from prior work, TSGs contain a multitude of components (commands, queries, natural language), each with a large number of input variations. Using PROSE [47] and its text transformation DSL (core to the FlashFill system), we show that even under such conditions, robust parsers can be learnt using a minimal set of input-output specifications.

6 DISCUSSION AND CONCLUSION

In this work, we presented a large-scale empirical study of over 4K+ TSGs mapped to 1000s of incidents. Our analysis indicates that TSGs are very frequently used for incident mitigation and notably help reduce mitigation time and effort. However, on studying feedback provided by 400+ on-call engineers at Microsoft, we uncover significant gaps in TSG quality, such as completeness, maintainability, readability, etc., characterized by our proposed taxonomy of issues. These insights motivate us to investigate the automation of TSGs and propose AutoTSG - a novel framework to aid with the automation of manual TSGs to executable workflows combining machine learning and program synthesis. Our evaluation of AutoTSG on 50 TSGs shows the effectiveness of the tool to both identify TSG statements (accuracy 0.89) and parse them for execution (precision 0.94 and recall 0.91). Lastly, with a survey of engineers at Microsoft, we show the usefulness of TSG automation and AutoTSG for TSG users. As next step, we are planning to deploy AutoTSG as a self-serve tool at Microsoft with an accompanying user interface and a feedback loop. We envision TSG authors uploading their current TSGs and viewing the automated TSG returned by our tool. Here, the author would edit/fix errors in the results, which we collect as feedback for re-training and improvement. Further, prior research [36] has shown that due to highly complex dependencies between services, on-call engineers find it challenging to mitigate incidents. With

AutoTSG and automated TSGs, we plan to automatically infer of a queue of TSGs to run, from various services, and help mitigate such complex incidents.

7 ACKNOWLEDGEMENTS

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C Shepherd. 2020. Software documentation: the practitioners' perspective. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 590–601.

[2] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1199–1210.

[3] Han Altae-Tran, Bharath Ramsundar, Aneesh S Pappu, and Vijay Pande. 2017. Low data drug discovery with one-shot learning. *ACS central science* 3, 4 (2017), 283–293.

[4] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[5] Jesper Andersen and Julia L Lawall. 2010. Generic patch inference. *Automated software engineering* 17, 2 (2010), 119–148.

[6] Chetan Bansal, Sundararajan Renganathan, Ashima Asudani, Olivier Midy, and Mathru Janakiraman. 2020. DeCaf: Diagnosing and Triaging Performance Issues in Large-Scale Cloud Services. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[7] Y-Lan Boureau, Jean Ponce, and Yann LeCun. 2010. A Theoretical Analysis of Feature Pooling in Visual Recognition. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (Haifa, Israel) *(ICML'10)*. Omnipress, Madison, WI, USA, 111–118.

[8] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[9] J. Chen, X. He, Q. Lin, Y. Xu, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. 2019. An Empirical Investigation of Incident Triage for Online Service Systems. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120.

[10] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang. 2019. Continuous Incident Triage for Large-Scale Online Service Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 364–375.

[11] Jie-Cherng Chen and Sun-Jen Huang. 2009. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software* 82, 6 (2009), 981–992.

[12] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[13] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 275–284.

[14] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. 68–75.

[15] Yan Duan, Marcin Andrychowicz, Bradly Stadie, OpenAI Jonathan Ho, Jonas Schneider, Ilya Sutskever, Pieter Abbeel, and Wojciech Zaremba. 2017. One-shot imitation learning. *Advances in neural information processing systems* 30 (2017).

[16] Li Fei-Fei, Rob Fergus, and Pietro Perona. 2006. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence* 28, 4 (2006), 594–611.

[17] Michael Fink. 2004. Object classification from a single example utilizing class relevance metrics. *Advances in neural information processing systems* 17 (2004).

[18] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*. PMLR, 1126–1135.

[19] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. 2013. Evaluating usage and quality of technical software documentation: an empirical study. In *Proceedings of the 17th international conference on evaluation and assessment in software engineering*. 24–35.

[20] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[21] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[22] Zellig S Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162.

[23] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*. 117–125.

[24] Jiajun Jiang, Weihai Lu, Junjie Chen, Qingwei Lin, Pu Zhao, Yu Kang, Hongyu Zhang, Yingfei Xiong, Feng Gao, Zhangwei Xu, et al. 2020. How to mitigate the incident? an effective troubleshooting guide recommendation technique for online service systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1410–1420.

[25] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 320–323.

[26] Shinji Kikuchi. 2015. Prediction of workloads in incident management based on incident ticket updating history. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 333–340.

[27] Nikita Kitaev, Steven Cao, and Dan Klein. 2019. Multilingual Constituency Parsing with Self-Attention and Pre-Training. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3499–3505. https://doi.org/10.18653/v1/P19-1340

[28] Gregory Koch, Richard Zemel, Ruslan Salakhutdinov, et al. 2015. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, Vol. 2. Lille, 0.

[29] Rrezarta Krasniqi, Siyuan Jiang, and Collin McMillan. 2017. Tracelab components for generating extractive summaries of user stories. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 658–658.

[30] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226* (2018).

[31] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[32] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting Neural Text Classification for Improved Software Categorization. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 461–472. https://doi.org/10.1109/ICSME.2018.00056

[33] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[34] Olaf Leßenich, Sven Apel, and Christian Lengauer. 2015. Balancing precision and performance in structured merge. *Automated Software Engineering* 22, 3 (2015), 367–397.

[35] Jing Li, Aixin Sun, and Zhenchang Xing. 2018. Learning to answer programming questions with software documentation through social context embedding. *Information Sciences* 448 (2018), 36–52.

[36] Liqun Li, Xu Zhang, Xin Zhao, Hongyu Zhang, Yu Kang, Pu Zhao, Bo Qiao, Shilin He, Pochian Lee, Jeffrey Sun, et al. 2021. Fighting the Fog of War: Automated Incident Detection for Cloud Systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 131–146.

[37] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[38] Rafael Lotufo, Zeeshan Malik, and Krzysztof Czarnecki. 2015. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Software Engineering* 20, 2 (2015), 516–548.

[39] Chen Luo, Jian-Guang Lou, Qingwei Lin, Qiang Fu, Rui Ding, Dongmei Zhang, and Zhe Wang. 2014. Correlating events with time series for incident diagnosis. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1583–1592.

[40] John MacFarlane. [n.d.]. Pandoc. https://pandoc.org/index.html.

[41] Senthil Mani, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey. 2012. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[42] Paul W McBurney and Collin McMillan. 2015. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.

[43] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices* 46, 6 (2011), 329–342.

[44] Microsoft. [n.d.]. "Azure Data Factory". https://azure.microsoft.com/en-in/services/data-factory/.

[45] Microsoft. [n.d.]. "Azure Monitor". https://docs.microsoft.com/en-us/azure/azure-monitor/overview.

[46] Microsoft. [n.d.]. "Kusto Query Language (KQL)". https://docs.microsoft.com/en-us/connectors/kusto/.

[47] Microsoft. [n.d.]. "Microsoft program synthesis using examples (prose) sdk.". https://www.microsoft.com/en-us/research/group/prose/. Accessed: 2022-05-19.

[48] Microsoft. [n.d.]. "Powershell". https://docs.microsoft.com/en-us/powershell/.

[49] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[50] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How can I use this method?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 880–890.

[51] Vinod Nair, Ameya Raul, Shwetabh Khanduja, Vikas Bahirwani, Qihong Shao, Sundararajan Sellamanickam, Sathiya Keerthi, Steve Herbert, and Sudheer Dhulipalla. 2015. Learning a hierarchical monitoring system for detecting and diagnosing service issues. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2029–2038.

[52] Rangeet Pan, Vu Le, Nachiappan Nagappan, Sumit Gulwani, Shuvendu Lahiri, and Mike Kaufman. 2021. Can program synthesis be used to learn merge conflict resolutions? an empirical analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 785–796.

[53] Constituency Parsing. 2009. Speech and language processing. (2009).

[54] Reinhold Plösch, Andreas Dautovic, and Matthias Saft. 2014. The value of software documentation quality. In *2014 14th International Conference on Quality Software*. IEEE, 333–342.

[55] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, Valletta, Malta, 45–50. http://is.muni.cz/publication/884893/en.

[56] John Robinson. 2014. *Likert Scale*. Springer Netherlands, Dordrecht, 3620–3621. https://doi.org/10.1007/978-94-007-0753-5_1654

[57] Amrita Saha and Steven CH Hoi. 2022. Mining Root Cause Knowledge from Cloud Service Incident Investigations for AIOps. *arXiv preprint arXiv:2204.11598* (2022).

[58] Jürgen Schmidhuber. 1987. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. Ph.D. Dissertation. Technische Universität München.

[59] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, and Nachiappan Nagappan. 2021. SoftNER: Mining Knowledge Graphs From Cloud Incidents. https://doi.org/10.48550/ARXIV.2101.05961

[60] Manish Shetty, Chetan Bansal, Sumit Kumar, Nikitha Rao, Nachiappan Nagappan, and Thomas Zimmermann. 2021. Neural knowledge extraction from cloud service incidents. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 218–227.

[61] Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical networks for few-shot learning. *Advances in neural information processing systems* 30 (2017).

[62] Jeffrey Stylos and Brad A Myers. 2006. Mica: A web-search tool for finding api components and examples. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 195–202.

[63] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip HS Torr, and Timothy M Hospedales. 2018. Learning to compare: Relation network for few-shot learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1199–1208.

[64] Yonglong Tian, Yue Wang, Dilip Krishnan, Joshua B Tenenbaum, and Phillip Isola. 2020. Rethinking few-shot image classification: a good embedding is all you need?. In *European Conference on Computer Vision*. Springer, 266–282.

[65] Secil Ugurel, Robert Krovetz, and C. Lee Giles. 2002. What's the Code? Automatic Classification of Source Code Archives. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Edmonton, Alberta, Canada) *(KDD '02)*. Association for Computing Machinery, New York, NY, USA, 632–638. https://doi.org/10.1145/775047.775141

[66] Amrisha Vaish, Tobias Grossmann, and Amanda L Woodward. 2008. Not all emotions are created equal: the negativity bias in social-emotional development. *Psychological bulletin* 134 3 (2008), 383–403.

[67] Joaquin Vanschoren. 2018. Meta-learning: A survey. *arXiv preprint arXiv:1810.03548* (2018).

[68] 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).

[69] Chi Zhang, Yujun Cai, Guosheng Lin, and Chunhua Shen. 2020. Deepemd: Few-shot image classification with differentiable earth mover's distance and structured classifiers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12203–12213.