# vBERT in Production: Nimbus Error Classification

## *An NLP-based Solution to Support Bundle Analysis*

Anonymous due to double blind review process


## ABSTRACT

In 2020, there were 2.5 Million VM deployment failures and 250K testbed deployment failures in Nimbus, representing 1.5% and 4.5% of each deployment type, respectively. To scale out the error triage process, we introduce an end-to-end machine learning system with two levels of error classification: L1 to determine the origin of the error [infrastructure, Nimbus, or product] and L2 [if it's a product problem] determine the Bugzilla Product, Category, and Component (PCC) classification by analyzing the log files from a VM support bundle. We demonstrate an NLP-based system capable of detecting deployment failures, extracting support bundles, determining the top-n candidate ERROR lines, preprocessing those error lines with vNLP, and use a fine-tuned vBERT model achieving an F-Score of 0.9788 to classify the Category and Component for trend tracking and automatic ticket filing in Bugzilla.


## 1. INTRODUCTION

Nimbus is R&D's internal cloud, composed of more than 100 vSphere Compute Centers and 1K physical ESXi hosts. Nimbus enables R&D teams to develop and test key VMware products including ESXi, vCenter, vSAN, NSX, and more. Each day, product teams deploy more than 200K VMs and more than 5K testbeds. As infrastructure demands grow each year, the Nimbus team continues to increase the number of physical hosts on a monthly basis to meet the needs of developers. But, with increased utilization comes an increase in the absolute volume of failures. In 2020, approximately 1.5% of VM deployments failed and 4.5% of testbed deployments failed. While the percentages of failures remain consistent on a quarterly basis, the increasing absolute volume of failures presents significant challenges to today's manual processes of triaging and root-causing each one.

We analyzed a substantial volume of deployment failures in Nimbus and identified duplicate deployment failures and incorrect bug assignment in Bugzilla as major areas for improvement. In 2020, there were a total of 25K Bugzilla tickets created for 250K testbed failures. For each ticket, at least three testbed deployment failures occur – one for the first occurrence, one for reproducing the failure, and one for debugging, which totals at least 75K testbed deployment failures to support the 25K tickets in 2020. As a result, the remaining 175K testbed deployment failures were duplicate failures that could have been avoided. On average, each testbed contains 10 VMs with 4 CPUs and is deployed for 30 minutes. Altogether, preventable duplicate deployment failures account for approximately 3.5 Million CPU hours (175K × 10 × 4 × 0.5). With each Nimbus Pod containing 12 Physical ESXi Hosts and each host having 26 CPUs with an average utilization of 70%, a single Nimbus Pod emits 150K pounds of CO2 per month [10]. As such, the overall sustainability impact for duplicate deployments is **3.3 Million pounds of CO2 per year**, which is equivalent to 69K mature trees.

To calculate the time lost due to misclassification in Bugzilla, we compared the average closing time for ESXi and vCenter tickets in 2020 against their toss rate, the number of times a ticket has its PCC reassigned. As shown in Table 1, each incorrect assignment will have a significant impact on engineering productivity with an added cost measured in days to close the ticket. After summing all of the lost time from incorrect assignment actions, **23,448 days were lost tossing tickets**. The duplicate deployment failures and toss rate slow down development and increase cost.

| Toss Rate (# of reassignments) | # of Tickets | Average +Days to Close Ticket | Days Lost |
|---|---|---|---|
| 1x toss | 3480 | +2.98 | 10,361 |
| 2x toss | 1146 | +4.26 | 4,885 |
| 3x toss | 484 | +3.74 | 1,812 |
| 4x toss | 263 | +5.79 | 1,523 |
| 5x toss | 156 | +9.44 | 1,473 |
| >5x toss | 228 | +14.88 | 3,392 |
| | | **Total:** | **23,448** |

Table 1: The cost of tossing Bugzilla tickets, measured in days.

The current processes for determining the cause of a deployment failure are manual. They require significant experience with Nimbus and specific components of a product and can result in duplicate failures and incorrect assignment. We present a system to replace those manual processes by consuming the inputs of a deployment failure and producing the outputs automatically.

A first round of analysis (L1) takes the error message raised in Nimbus logs and determines whether it occurred in the underlying infrastructure, Nimbus's own server, or due to a product specific bug or misconfiguration. Infrastructure and Nimbus-specific issues have well-documented solutions and processes, which remain in place. Product-related failures are more complex and require additional classification (L2) to determine the likely issue and the correct team to act. When a Nimbus deployment fails for a product-related reason, a support bundle is generated containing tens to hundreds of thousands of log lines from the target VM's syslog. Manual analysis of these logs is not feasible. Although services like Log Insight make it easier to dissect the logs, significant engineering time is still required to identify the issue.

To address the challenges in L2, we introduce a novel Natural Language Processing (NLP) pipeline. To train this system, we sourced support bundles from resolved bugs in Bugzilla and a complement of support bundles from successful Nimbus deployments. We first filter the support bundle to only ERROR lines and then determine the top-10 error lines. Those lines are then preprocessed with VMware's Natural Language Preprocessor (vNLP) to remove noise and split compound words. The resulting string is then fed into our custom language model, vBERT, to classify the Category and Component for trend tracking and automatic filing in Bugzilla.

## 2. APPROACH

Support bundles are the primary source of truth for diagnosing and triaging a troubled system. Despite being generated by computers, the logs contained in a support bundle are difficult to parse, as they are not output in a machine-readable format, but rather in a pseudo-human-readable format. As such, training either a machine learning model (or a human support engineer) to read them is challenging.

When considering how to ingest such content in a machine learning model, there are two primary methods for considering the data: either as a finite set of lines, or as text. Both are valid and have their drawbacks. Let's first consider them as a finite set of lines. We gathered over 100 support bundles, concatenated all of the log files, and removed duplicate lines. The original several million lines were reduced to approximately 50K unique lines, this being the theoretical finite set of log lines. Each support bundle is then transformed into a single, one-hot vector [6] of the lines found within it. That vector is suitable as input to a machine learning model. There are three major drawbacks to this approach. First, the quality and utility of the resulting finite set is limited by our ability to correctly determine whether any two lines are duplicates or unique, i.e., were they preprocessed correctly such that only noise and no pertinent information was removed. Second, there is no ability to handle previously unseen lines, i.e., unless the analyzed set of support bundles contains all possible log lines (unlikely), any previously unseen lines would be dropped, regardless of their importance to the root cause. And, lastly, machine learning models generally achieve better performance with dense vectors where most elements are non-zero compared to sparse vectors. A 50K-length vector comprised of mostly zeroes is extremely sparse.

Let's next consider a support bundle as text instead of as a finite set of lines. In this case, the above listed problems are all solved, but with its own drawback, that being the needle-in-a-haystack problem: support bundles can have hundreds of thousands of log lines, but only a small handful of lines are relevant to the root cause of the problem. Modern language models are extremely proficient at classifying text up to the model's maximum sequence length (often only a few hundred tokens[1]). Thus, we are left with a choice. Either we'd need to find a method of breaking a support bundle into chunks of text that fit in a model's maximum sequence length, generating a vector that represents each chunk, and combining all those vectors into a single vector that represents the entire bundle, or determine the small handful of lines relevant to the root cause and use them as the input to the model. The first option is akin to encoding the entire haystack and hoping that the needle sticks out, while the second approach searches for the needle and only encodes that, throwing away the haystack in the process. We opted for the second approach.

Finally, one must determine the classes for classification. The original proposition was to perform unsupervised clustering to see if there were any patterns to the failures. While feasible, this leaves the unenvious task of deducing the label for each group. Thus, we pivoted to using Bugzilla Categories and Components as labels so we could use supervised classification.

### 2.1 Root Cause Identification

Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical method for identifying words that are important to a document in a corpus. This concept can also work for identifying important log lines within a support bundle, i.e., determining the top-n ERROR lines within a support bundle [1]. A collection of normal support bundles forms the basis of the corpus. In a support bundle from a failed deployment, the lines which are "most deviating" are thus determined to most likely be related to the root cause of the issue.

While it is challenging to directly assess the quality of the selected lines (it would require expert-level review of each support bundle and the selected lines), it can be assessed via a meta measure, i.e., if classification achieves a high F-score using the selected lines, it can be safely assumed that the chosen lines were, in fact, highly correlated with the root cause.

### 2.2 Modeling the Language of VMware

Language modeling is a method of generating semantically meaningful vector representations for sequences of words. Released by Google in 2018, BERT: Bidirectional Encoder Representations from Transformers [2] made significant improvements to the state-of-the-art in language modeling and NLP in general. Attempts to benchmark BERT on VMware-specific tasks showed improvements over previous approaches but fell short compared to the staggering improvements demonstrated on public benchmarks. A deep-dive analysis revealed a pervasive out-of-vocabulary (OOV) problem to be the cause of failure. When BERT encounters a word that is not in its vocabulary, BERT breaks the word down into a root token and known sub tokens, e.g., "colorless" becomes [ 'color', '##less' ]. This approach works well for common suffixes. Reducing the vocabulary allows the language model to learn more robust representations of each token. However, this approach fails for product names, technical jargon, and compound words, e.g., "vSphere" becomes [ 'vs', '##pher', '##e' ], "Kubernetes" becomes [ 'ku', '##ber', '##net', '##es' ], "acceptunverifiedcertificates" becomes [ 'accept', '##un', '##ver', '##ified', '##cer', '##ti', '##fi', '##cate', '##s' ]. No additional understanding is gained by those splits. The solution to this problem is to train BERT on the language of VMware.

To do so, one must first preprocess our content, such as Documentation, KB articles, etc., to prepare it for ingestion. vNLP [3], was specifically designed to preprocess our content for ingestion into a machine learning model. vNLP is a pipeline of preprocessing stages that fixes encoding issues, strips repeated

[1] A token is a word or sub-word that appears in a language model's vocabulary. Words which are out of vocabulary may either be dropped or split into a sequence of tokens.

characters, filters noise such as timestamps and hex strings, fixes spelling mistakes, splits compound words, and expands contracted words, all with the goal of reducing token count and solving the OOV problem. Preprocessing our content and pre-training BERT yields a custom version, vBERT, which outperforms BERT on VMware-specific tasks [4][5]. The corpus used to pre-train vBERT contains a considerable volume of log content and is thus well suited to classifying log lines.

## 3. DESIGN

When a Nimbus deployment failure occurs, the automated triage flow shown in Figure 1 is followed.

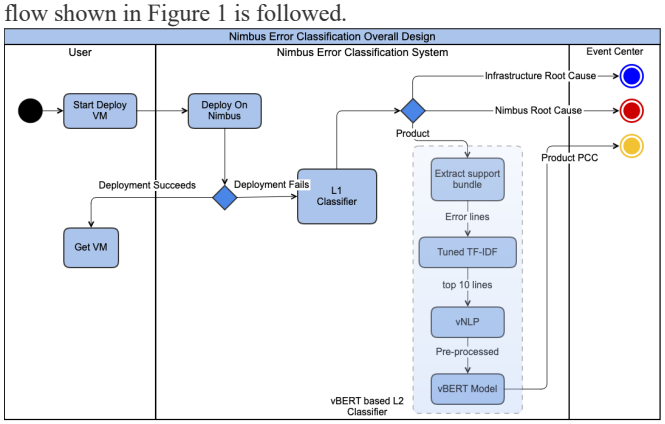


Figure 1: Diagram of overall system flow.

### 3.1 L1: Failure Type Determination

L1 is a lightweight first-hand router. For a deployment failure, error messages raised by Nimbus are sufficient to determine high-level root causes such as a parameter error, insufficient Nimbus resources, etc. L1, is a set of regular expressions. The regular expressions were generated automatically based on historical error messages [9]. To generate the regular expressions that comprise L1, log lines were clustered based on similarity. This was accomplished with Doc2Vec [7] to vectorize the log lines and DBSCAN [8] to cluster them with labels assigned to each cluster by manually reviewing them.

### 3.2 L2: Nimbus support bundle classification

L2 intakes a support bundle from a failed deployment and classifies according to Bugzilla PCC. The three key components of L2 are the top-10 error line selector, vNLP, and vBERT.

#### 3.2.1 Choosing the Top-10 Error Lines

We modified the TF-IDF approach found in the 2019 NSX RADIO paper [1] to calculate the top 10 most important error lines for a given support bundle. Successful support bundles are used to compute the IDF value while support bundles from failed deployments are used to compute the TF value. In our implementation, only error messages are considered for both TF and IDF as they are more likely to be correlated with the root cause of a failure. This approach reduces the input from tens or hundreds of thousands of lines to 10 lines of high importance, which fit in the input limit of the vBERT classifier after being cleansed with vNLP.

#### 3.2.2 Input Cleansing with vNLP

By using vNLP to preprocess the log files, we were able to reduce noise and token count. For example, the following log line:

```
2018-05-30T10:09:35.326Z cpu33:67219)DOM:
DOMOwner_SetLivenessState:4898: Object
f513635a-e6ab-ae78-f497-000f5355c750 lost
liveness [0x439de26c0c80]
```

is cleansed to:

```
timestamp cpu dom dom owner set liveness
state object hex id lost liveness hex id
```

The original line is 79 tokens long while the cleansed line is only 19 tokens long. This significant reduction in token count was critical in getting the error lines to fit inside vBERT's maximum sequence length of 512 tokens.

#### 3.2.3 vBERT-based Classifier

vBERT itself is only a language model. To use it for classification, a single, fully connected layer was added with the input width being equal to the output width of the last Transformer [14] layer of the language model, and output width being the number of classes. Cross entropy [12] is the loss function and ADAM [13] is the optimizer. Despite PCC being inherently hierarchical, there was no need to implement hierarchical classification. The output class is simply the combination of category and component.

## 4. EXPERIMENTAL DESIGN

As with any machine learning project, there is a large set of hyperparameters that must be tuned to achieve an ideal result. Given our highly imbalanced dataset, one of the most important considerations was to determine if some of the minority classes needed to be dropped. Modern language models can achieve robust results with just a few dozen examples compared to the need for hundreds or thousands of training examples for older models. However, there is still a floor for useful results[2]. Our dataset has a range of example counts per class from over a thousand down to just 1. As such, a cutoff must be determined for inclusion in the training set.

| Minimum Bundles per Class | F-Score | % of Dataset |
|---|---|---|
| 100 | 0.9984 | 79.6% |
| 50 | 0.9762 | 90.3% |
| **20** | **0.9591** | **94.6%** |
| 5 | 0.9141 | 98.9% |

Table 2: F-Score vs Class Cutoff (initial investigation)

[2] Massive state-of-the-art language models have demonstrated a limited ability to perform few-shot or single-shot learning, but reasonably sized models are not there yet.

We evaluated the tradeoffs between F-Score and dataset coverage and chose a class cutoff of 20 as seen in Table 2. The set of retained support bundles represents 95% of the total dataset. Consequently, mislabeled bugs will need to be tossed.

We also experimented with grouping support bundles by Category for those that fell below the 20-bundle threshold when split by Category and Component to see if we could include additional bundles in the training set. Unfortunately, no groups could be added without dropping the F-Score below 0.95. This was not an unexpected result, as machine learning models generally don't respond well to grouping disparate items, since interpolating between the disparate groups can cause confusion.

## 5. MODEL EVALUATION

Initial models were evaluated with a 10% evaluation set and a 10% test set. We performed stratified and nested K-Fold Cross Validation to analyze the generalizability of the model, with K=5 and the random seed being the nested hyperparameter.

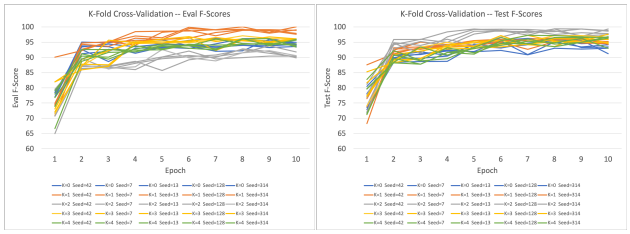


Figure 2: K-Fold Cross Validation graphs for the evaluation (left) and test (right) sets.

As can be seen by the trend in Figure 2, the model does generalize well, but the effects of the smaller classes on the data splits is also apparent. With an 80/10/10 split, the minority classes have as little as 16/2/2 examples in each set, thus the apparent discrepancy between the orange (K=1) and grey (K=2) sets in the eval and test sets. Because of the variability introduced by the minority classes and because the model did appear to generalize well, we decided to combine the eval and test sets into a single eval set.

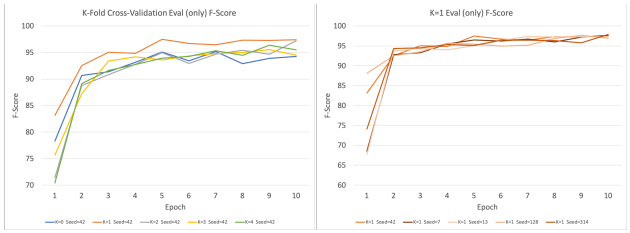


Figure 3: K-Fold Cross Validation of the combined eval/test sets.

As shown in Figure 3, only doing a single 80/20 split, made it easier to choose a set of hyperparameters for optimal training. An additional round of K-Fold Cross-Validation with the new data split yielded optimal values for our hyperparameters and ultimately achieved a final F-Score of **0.9788**.

## 6. DEPLOYING TO PRODUCTION

With newly added support for BERT-based models like vBERT, we leveraged InstaML's [11] infrastructure and APIs to tune hyperparameters, perform training, and evaluate candidate models while iteratively cleansing the input data. The model with the best F-Score is automatically selected for deployment into the inference pipeline. There are two options for deployment from InstaML: allow InstaML to host the model for inference or download the lightweight microservice template code in a docker container for self-hosting. For this case, we chose to make use of the docker container so that the whole system could run inside Nimbus.

## 7. FUTURE WORK

With the experimental value of this approach proven and an implementation for vCenter in production, our team will focus on expanding to cover additional products and add further automation to trigger events based on error classification. The next target is to reproduce these results for ESX. This will require computing TF-IDF values specifically for the ESX log lines and fine-tuning a new vBERT-based classifier, but the overall process will remain the same.

To increase the coverage of PCC in the training set, i.e., to assist with the minority class/data imbalance problem, we are also building a new system to collect bundles directly from CAT (See Appendix B for additional information). If there is a bundle mentioned in a ticket, it will be saved automatically and used during a future model refresh.

Additionally, we're applying a similar NLP-based approach to triage Nimbus test launcher post-boot failure logs.

## 8. CONCLUSION

With a final F-Score of 0.9788, the combination of TF-IDF for top-n error line selection, data cleansing with vNLP, and classification with our VMware-specific language model vBERT, clearly demonstrates the effectiveness of using an NLP-based approach for working with log data. Thus, with the first model covering vCenter in production, our immediate next steps are to work with CPBU QE teams to measure engineering productivity and sustainability gains, and then duplicate the results with ESX data.

Additionally, we see opportunity to integrate this solution into the Customer Experience & Success support workflow to provide a support bundle analysis service for TSEs by promoting the top error lines detected by this system. Moreover, with the increasing adoption of Skyline, there are likely components of this system that could be used to analyze specific topologies from Skyline customers that match Nimbus topologies that resulted in failures.

## 9. ACKNOWLEDGEMENTS

This cross-team effort relied on the innovative spirit, expertise, and hard work of many team members. We are grateful for the support from all who assisted in this effort.

## 10. REFERENCES


[1] Kate Zhang, et al (2019), Automatic Root Cause Identification: A Term Frequency – Inverse Document Frequency Recomposition Based Approach. https://radio.eng.vmware.com/2019/event_files/11711/download.

[2] Jacob Devlin, et al (2018), BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. https://arxiv.org/pdf/1810.04805.pdf.

[3] Rick Battle (2020), vNLP: VMware's Natural Language Preprocessor. https://vbetter.vmware.com/learn/5f99b8e2e82a9afff5c289b3.

[4] Rick Battle (2020), vBERT: Initial Results Pre-training BERT on VMware Content. https://vbetter.vmware.com/learn/5ed58e4b9f0d99140642eac7.

[5] Rick Battle (2020), vBERT 2020: Release Announcement. https://vbetter.vmware.com/learn/5fa34942565942faa28b0cf6.

[6] Matthew Mayo (2018), Data Representation for Natural Language Processing Tasks. https://www.kdnuggets.com/2018/11/data-representation-natural-language-processing.html.

[7] Quoc Le, et al (2014), Distributed Representations of Sentences and Documents. https://arxiv.org/pdf/1405.4053.pdf.

[8] Martin Ester, et al (1996), A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf.

[9] Pinjia He, et al (2017), Drain: An Online Log Parsing Approach with Fixed Depth Tree. https://pinjiahe.github.io/papers/ICWS17.pdf.

[10] Yugansh Aggarwal (2019), How Green is Nimbus. https://video.eng.vmware.com/videos/ae7b333b-89cc-45d5-9af3-74c45dc201ba.

[11] Steve Liang, et all (2018), InstaML: Highly Performant Automated Machine Learning. https://radio.eng.vmware.com/2019/event_files/11851/download.

[12] Jason Brownlee (2019), A Gentle Introduction to Cross-Entropy for Machine Learning. https://machinelearningmastery.com/cross-entropy-for-machine-learning/.

[13] Diederik Kingma (2014), ADAM: A Method for Stochastic Optimization. https://arxiv.org/pdf/1412.6980.pdf.

[14] Ashish Vaswani, et al (2017), Attention Is All You Need. https://arxiv.org/pdf/1706.03762.pdf.


## Appendix A: Architecture

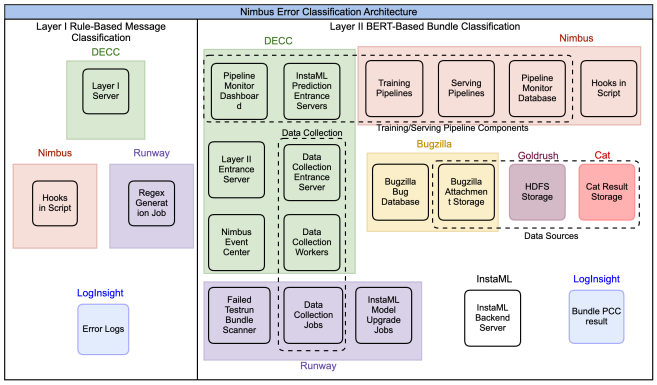


Figure 4: Diagram of system components and hosting infrastructure.

For L1, the classification server is deployed as a container in DECC. The rule generation job is deployed in Runway, and the historical error logs are sourced from LogInsight. Hooks are added in nine main deployments, such as vcvadeploy, esxdeploy and ovfdeploy.

For L2, things are more complex. We use DECC for entrance servers and some backend services. We use Runway for cronjobs and system CI. We use a Nimbus Long-run Pod for heavy usage data processing jobs. Lastly, we source bundles from Bugzilla, Goldrush, and Cat.

## Appendix B: Data Collection

| From 2018-01-01 | ESXi | VC |
|---|---|---|
| PRs | 51523 | 18147 |
| PRs attached bundles | 153 | 152 |
| Nimbus Result folders attached | 337 | 4261 |
| all bundles attached | 10096 | 6316 |
| bundles final | 1776 | 3681 |
| PRs final | 39 | 94 |

Table 3: Assessment of support bundle counts and their associated bugs.

The support bundles we used for L2 training are from Bugzilla from 2018-01-01 to 2020-10-01. We chose closed PRs under product ESXi and vCenter. After we got those PR numbers, we scanned the Bugzilla attachment storage to find a Nimbus results folder. Support Bundles are collected with a Nimbus script when the testbed deployment has failed in a specific name format. Based on that, the support bundles could be found. Usually for a testbed deployment, there will be more than one ESXi host that gets deployed. Because of this, there will be more than one support bundle in the Nimbus results folder for ESXi PRs. In order to find the correct ESXi support bundle, several rules are implemented, e.g., the closer the ESXi name is to the error line in Nimbus logs the higher score it will achieve. We select the ESXi support bundle with the highest score. As a result, we collected 1,776 ESXi support bundles and 3,681 vCenter support bundles for training.

| From 2018-01-01 | ESXi | VC |
|---|---|---|
| PR closed | 15106 | 5715 |
| PR closed whose comments has keyword "bundle" | 3318 | 1276 |
| comments mentioned keyword "bundle" | 9630 | 3325 |

Table 4: Assessment of Bugzilla tickets and comments containing the keyword “bundle”.

During the data collection, we found that the support bundles we collected were from a very small number of PRs. The negative impact is that we have to drop some PCC due to not meeting the ≥20 bundles threshold for inclusion. Additional analysis revealed that the keyword “bundle” was mentioned in 1/5 of the PRs. In order to leverage the efforts for first-hand analysis of PRs and also increase the coverage of PCC, we are building a new system to collect bundles directly from CAT. Once there is a bundle mentioned in a PR, it will be saved automatically.