2024 IEEE International Conference on Cloud Engineering (IC2E) | 979-8-3315-2869-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/IC2E61754.2024.00027

PraxiPaaS: A Decomposable Machine Learning System for Efficient Container Package Discovery

Zongshun Zhang*, Rohan Kumar*, Jason Li*, Lisa Korver*,
Anthony Byrne**, Gianluca Stringhini*, Ibrahim Matta*, Ayse Coskun*
Boston University, Red Hat, Inc.***

***Abstract*—Due to the increasing complexity of cloud architectures, automatically tracking and inspecting container packages in Platform-as-a-Service (PaaS) clusters are challenging tasks. This introspection capability, however, is critical to identify vulnerable packages and compile an accurate Software Bill of Materials (SBOM). Motivated by introspection frameworks focusing on virtual machine (VM) settings and ML methods for software discovery, we design *PraxiPaaS* as a framework to inspect PaaS container images with a highly scalable ML inference pipeline by scanning file changes during package installations. Our ML pipeline includes a *structured* collection of word2vec encoders and a corresponding *structured* ML model to achieve short incremental training time for incorporating additional packages while maintaining a high F1-score in generating the SBOM. Our evaluation shows that our structured ML pipeline provides an exponential drop in incremental training time from 2.8 hours to 8.6s with 32 CPU cores, while maintaining an F1-score of 0.82, compared to the traditional monolithic model design. We deploy a prototype of *PraxiPaaS* in the New England Research Cloud (NERC) OpenShift cluster and evaluate the inference time comparing structured versus monolithic model design.**

***Index Terms*—cloud computing, machine learning, model decomposition, PaaS, software discovery, SBOM**

I. Introduction

Platform-as-a-Service (PaaS) clusters, such as Kubernetes and OpenShift, have gained significant attention owing to the widespread adoption of microservices [1], [2]. However, as more users pull container images into a cluster, container image management has become increasingly labor-intensive and time-consuming. Developers often do not build their software stacks from scratch, resulting in open-source software projects that rely on many other existing projects [3]. Furthermore, software components may be introduced, modified, or removed at different frequencies. Such dynamic behavior adds substantial operational overhead when keeping track of vulnerabilities, performance, or compliance for software stacks. Cluster administrators may wish to monitor the Software Bill of Materials (SBOM) [3], [4] of service containers so that they can promptly react to "anomalies", ideally without depending on information provided by tenants. For example, especially in academic clusters such as the New England Research Cloud (NERC) [5], outdated or vulnerable packages might be present because tenants focus more on efficiency or quick delivery of their implementation due to the pioneering nature of their work.

To ensure that only vulnerability-free and compliant packages are running on a cluster and also to provide a container package log to help analyze potential anomalies or failures in the future, a naïve method is to maintain a list of all dependencies. However, such a list is fragile and difficult to maintain due to daily changes in software. Thus, automating such steps and developing a framework to track and log software packages in PaaS containers have become urgent needs as today's software projects scale up quickly [6].

Some software discovery methods create predefined *rules* with expert inputs. For example, when the python package "fiona" got upgraded from version "1.9.4" to "1.9.5", it updated its library "libcurl-750590ea.so.4.7.0" to "libcurl-fiona-4ac9f96f.so.4.8.0" in order to fix a critical vulnerability in "libcurl" [7]. Cluster administrators can create a specific rule to distinguish versions of "fiona", and alert users to update the package or proactively upgrade the package for the users. Such alerts can be particularly valuable for packages like "fiona", mainly used by data scientists working with geographic data, as users may not prioritize patching security vulnerabilities. However, maintaining the rule generation pipeline is laborious and fragile. Some other tools query the package management installation database, e.g., *dpkg* [8], but these methods have shortcomings. Developers might not use package managers to install packages, e.g., when building with *CMake* or from a repository. Package managers are also vulnerable to attacks, e.g., man-in-the-middle attacks [9], which can compromise the integrity of installation database records.

Instead of relying on predefined rules with expert input or package installation DBs, recent efforts [10]–[12] use ML techniques to discover packages using fingerprints extracted from file system changes during installation, i.e., *discovery by example* [10]. Notably, these earlier methods raise training efficiency challenges, especially when incrementally adding new package (labels), due to their *monolithic* ML model design. In DeltaSherlock [11], the authors consider traditional word2vec [13] methods, which, when introducing new labels, require retraining the full word2vec encoder and model. When training an XGBoost model to discover $3,000$ packages using 32 CPU cores, DeltaSherlock requires approximately 2.56 hours of training time ($6.26 using AWS c6a.16xlarge instance in Fig. 2). This extremely long training time is unacceptable for popular (albeit possibly vulnerable) packages, as a large number of users may have downloaded them during that time. For example, considering only PyPI packages, the $8,000^{th}$ most downloaded package got downloaded $41,167$ times just in Jan. 2024 [14]. Furthermore, each package can have tens of versions and can get updated daily in a continuous integration

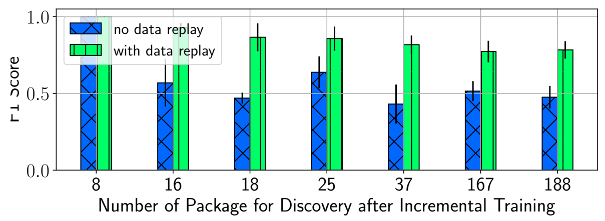


Fig. 1: During incremental training rounds (shown by the x-axis when adding new labels), Praxi requires model calibration (*data replay*) to maintain F1-score.

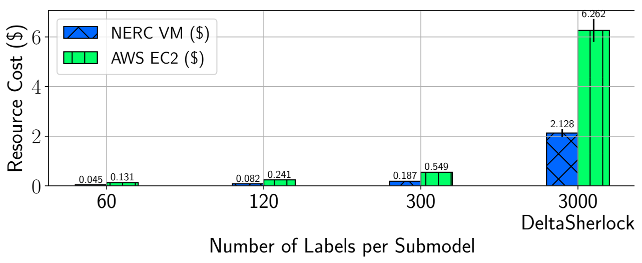


Fig. 2: A structured model is more cost-efficient than a monolithic model for SBOM generation as it enables incremental training. The costs are based on training time using a 32 CPU cores VM in NERC or a c6a.16xlarge instance in AWS.

and continuous delivery (CI/CD) pipeline [15].

Generally, the training cost of a monolithic model design increases quickly. Similar scalability issues are also observed in Praxi [12], which builds ensemble models using Vowpal Wabbit [16] for incremental training. For single package prediction tasks, Praxi proactively builds surplus classifiers for future package labels. When incrementally adding a new label, they fit an unassigned classifier to output high confidence for the new class and others to output low confidence. However, training unassigned classifiers wastes computation resources, and it requires retraining the complete model and adding new classifiers when the number of labels is above the number of classifiers. For multi-package prediction tasks, Praxi fits new classifiers for new labels, which requires full model retraining to calibrate classifiers [17] introduced in different incremental training rounds. In Fig. 1, we show the F1-score for predicting multiple packages from each installation fingerprint after incrementally adding packages for discovery in Praxi. Without model calibration (*no data replay*), the F1-score drops quickly. And with historical data replay, the F1-score can be maintained. Essentially, the training paradigm of Praxi is close to that of a monolithic model.

In this paper, we present *PraxiPaaS*[1], a framework for container package discovery based on package installation changes using a decomposable *structured* ML model design. Similar to Praxi and DeltaSherlock, we leverage the file changes during package installations as installation fingerprints and feed this information into our structured word2vec ML inference pipeline to predict the package(s) installed. In contrast to Praxi and DeltaSherlock, we solve key challenges in practical deployment and model scalability by loading installation fingerprints from container layers and introducing a structured word2vec ML method, motivated by Mixture of Expert ML systems [18], [19], to achieve high F1-score while mitigating the scalability drawbacks in the PaaS setting. Our *structured model* decomposes the traditional monolithic model along package labels. Each submodel in the structured model focuses on a subset of similar packages from the training dataset measured by pairwise cosine similarity in tokenized path names of files changed during installations.

Model decomposition allows for incrementally training and adding submodels, which minimizes redundant training and inference costs for similar packages, thereby overcoming the scalability issue of previous works. In Fig. 2, with $3,000$ packages, training costs per submodel drop exponentially as the number of package labels per submodel decreases. We evaluate training time in Sec. IV-C and inference time in Sec. IV-F.

Meanwhile, a *structured model* also introduces false positives. To maintain high precision and F1-score against previous works, PraxiPaaS proposes a package similarity-based submodeling approach (*Submodel Routing*). For example, the Python package "vine==5.0.0" dropped support for Python 3.4 and earlier, and removed the "backports" module compared to "vine==1.3.0". Since most file path names remain unchanged, naïvely assigning the two versions to different submodels can make it hard to classify the versions. Neither submodel can realize that the "backports" module can distinguish the two versions. Furthermore, the effort to train such submodels is redundant and wasted. Instead, retraining a submodel to add "vine==5.0.0" alongside with other similar packages ensures the model efficiently learns the differences in the "backports" module. In Sec. III-B, we detail our package similarity analysis and show that our structured model gives high precision after tuning a similarity threshold for submodel building.

Our main contributions are summarized as follows:

1) We design *PraxiPaaS*, a cloud operation pipeline to discover container software for vulnerability detection in PaaS clusters based on file changes during installations.
2) We design a highly scalable word2vec-based decomposable *structured* ML inference pipeline with our cosine similarity-based *Submodel Routing*. It adapts Mixture of Experts modeling [18], [19] to save resource cost.
3) We formulate an optimization problem to trade F1-score and training time, given input dimensions, labels per submodel, samples per label, package similarity threshold, and a filter for duplicate tokens among packages.
4) We deploy a prototype of *PraxiPaaS* in the NERC OpenShift cluster to generate container SBOMs using Red Hat OpenShift AI Pipelines [20] and Red Hat Advanced Cluster Management (ACM) [21].

Combining model decomposition and *Submodel Routing*,

[1]https://github.com/ai4cloudops/Praxi-Pipeline

our structured model is more scalable than monolithic models. *PraxiPaaS* provides 8.6 seconds per incremental training step and 0.82 F1-score. Meanwhile, model retraining takes $10,052$ $(110,681)$ seconds in DeltaSherlock (Praxi). By invoking a subset of submodels during inference, *PraxiPaaS* achieves 0.03s model inference time versus 14.63s for DeltaSherlock (192 images in a batch). We believe our contributions in building a decomposable ML system for the cloud will inspire other efficient, accurate, and scalable implementations of ML pipelines in the cloud.

The paper's organization is as follows. Section II presents the background and related work. We then introduce *PraxiPaaS* design in Section III. In Section IV, we evaluate the system scalability with respect to F1-score, training and inference time. Section V concludes the paper with a summary of contributions, possible extensions, and future work.

II. Related Work

We review previous works related to software discovery methods and frameworks, and compare them with our proposed system. We also discuss highly scalable ML models and system designs that motivate our ML-based framework.

A. Package Discovery Methods

Previous works propose several package discovery methods. Rule-based methods [22], [23] generate rules to match package with expert inputs, e.g., checking the existence of dependent libraries or executables with hashes, which is not scalable given continuous integration and continuous delivery (CI/CD) practices with frequent software updates. Other tools [8], [24]–[28] leverage installation DBs of package management systems, e.g., dpkg. However, they miss cases when users install packages with CMake or from a remote repository.

Praxi [12] and DeltaSherlock [10], [11] use ML models to automatically predict the labels of the packages installed, leveraging file system changes during installations as input features. However, Praxi, when predicting multiple packages from each installation fingerprint using Cost-Sensitive One Against All (CSOAA) reduction in Vowpal Wabbit, suffers from un-calibrated classifiers for each label after incremental training. The classifiers for new labels introduced during incremental training do not train with the historical data. Remedies include retraining all classifiers by replaying historical data or classifier re-weighting methods [17], [29], [30]. This drawback leads to limited scalability, and the training paradigm is more akin to a monolithic model design similar to DeltaSherlock, requiring complete model retraining. We instead propose to build independent submodels for different package labels so that we only calibrate classifiers inside each submodel, e.g., an ensemble model, and avoid calibration across submodels.

B. Package Discovery Frameworks

Orthogonal to package discovery methods, prior works also design practical frameworks to enable the deployment of software discovery services in large-scale clusters. Some works let users submitting images to an introspection service [31], [32] or the service proactively inspects a snapshot of the VM's remote disks [33] harnessing diskless provisioning methods [34], [35]. Our work, instead, is an automated introspection pipeline for containers in PaaS. *PraxiPaaS* automatically identifies container images used in the cluster through cluster observability, i.e., Red Hat Advance Cluster Management. Then, we pull image layers and generate the SBOM.

C. High Scalability Word2Vec ML Pipelines

Text summation methods [13], e.g., word2vec [36], map split texts, i.e., tokens, to a numeric space for measuring the distances between samples and classifications. We apply this idea to predict packages installed by measuring distances between an observed package installation fingerprint and packages in the training data. Previous work [11] uses a similar method. However, their method is not scalable when new features and new packages are added to the software discovery task, as they have to retrain the embedding mapping dictionary in word2vec and the inference model from scratch. To mitigate such overhead, we design a structured word2vec encoder and classifier method to enable incrementally adding or retraining individual submodels dedicated to discovering new packages.

D. High Scalability ML System Designs

Previous works focus on reducing redundant computations for ML models to enable large-scale training and inference while minimizing monetary resource cost. Model distillation [37], Deep Learning (DL) model hidden variable compression [38]–[40], and weights pruning [41]–[43] and skipping [44]–[47] are popular DL system adaptation methods for reducing memory, storage, and computation overheads. Within the domain of building a structured model, with constituent submodels focusing on sub-tasks, SplitNet [44] and other Mixture of Experts models [18], [19] split labels semantically among multiple DL models. For example, if we are already confident that a photo includes an animal, only the classifiers for different animals should be used downstream. We apply a similar concept and analysis based on the distribution of our installation fingerprint data.

We design a structured model with a collection of XGBoost submodels, where each submodel handles a unique subset of packages with high pairwise cosine similarities between installation fingerprints. In contrast to previous work, we partition labels and incrementally add packages for SBOM generation while preserving high scalability by leveraging the installation fingerprint data distribution.

Overall, we consider the existing spectrum of container package inspection methods and frameworks to be either not scalable to track rapidly growing software project dependencies or inefficient with container package discovery. So we develop *PraxiPaaS*, our container package discovery method using ML for PaaS clusters, motivated by previous word2vec methods and scalable ML system designs.

III. *PraxiPaaS*

PraxiPaaS is a container package discovery framework for PaaS clusters using ML techniques. As shown in Fig. 3, it

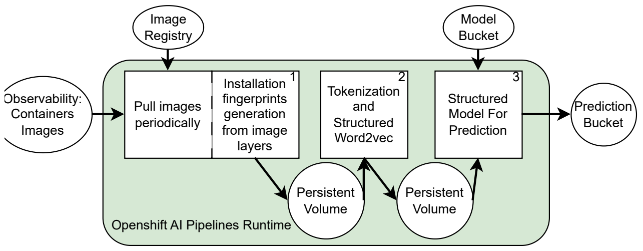


Image Registry
Model Bucket
Observability: Containers Images
Pull images periodically
Installation fingerprints generation from image layers
1
Tokenization and Structured Word2vec
2
Structured Model For Prediction
3
Prediction Bucket
Persistent Volume
Persistent Volume
Openshift AI Pipelines Runtime

Fig. 3: System Architecture of PraxiPaaS: It employs three steps using OpenShift AI Pipelines: (1) Pull container images based on cluster observability (Red Hat Advance Cluster Management) from an image registry (Docker Hub) and read file changes in image layers. (2) Tokenize the pathnames of files changed. (3) A structured model with XGBoost submodels tailored to package token distribution infers packages installed.

Tokens			Packages	
"__init__."	"res"	"Tiff"	boto3	pillow
13	45	0	1	0
3	0	6	0	1

TABLE I: A partial *token-frequency table* from the training dataset with two sample rows after vectorization: The first sample has tokens extracted from **boto3** installations, and the second has **pillow**. **"__init__."** can be found in many Python package installations, which can lead to false positive predictions by submodels. **"Tiff"** and **"res"** are the more distinguishable features to classify **boto3** and **pillow**.

has three components prototyped with Red Hat OpenShift AI Pipelines in the New England Research Cloud (NERC) cluster. The OpenShift AI Pipelines runtime provisions a pod for each component and several persistent volumes for communications between components. The first component retrieves the container image of interest from Docker Hub by reading the container images used in NERC from Red Hat Advanced Cluster Management (Prometheus & Grafana). It also loads the file change log of each container layer as the fingerprint of the package(s) installed. The second component uses a word2vec ML pipeline to tokenize the fingerprints and vectorizes the tokens of each sample using the *token-frequency table* encoding, popular in information extraction tasks [48]. The third one downloads a pre-trained structured model using XGBoost submodels from a storage, e.g., AWS S3, to predict the packages installed and writes the predictions to another storage unit. In the rest of this section, we discuss our design decisions and insights to configure the ML inference pipeline.

A. PraxiPaaS Architecture

We discuss installation logs retrieval in PaaS clusters (Sec. III-A1), structured word2vec encoder design (Sec. III-A2), and structured model design (Sec. III-A3), as outlined in Fig. 3.

1) File Changes Logs Extraction: PraxiPaaS leverages file changes during package installations to discover packages. In PaaS clusters, we monitor new container launches through various k8s, OpenShift, and RH Advanced Cluster Management metrics for a target namespace, e.g., **kube_pod_container_info**. Then we pull the container images from Docker Hub or other registries and read file changes by *layers* [49].

Similar to Praxi's changeset generation [12], we generate a fingerprint for each package installation using path names of modified files in each container layer. In this way, we never execute containers, which reduces resource demand and does not expose container security vulnerabilities during inspection.

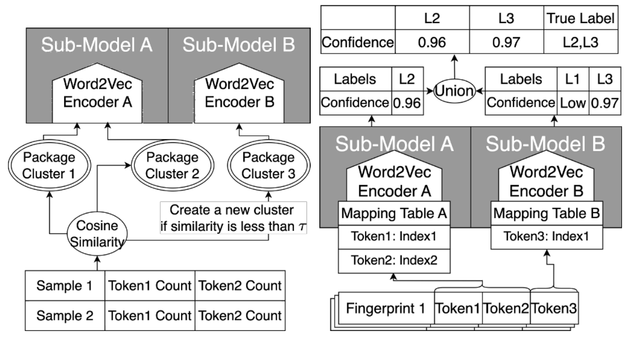
2) Structured Word2Vec: To achieve optimal precision, F1-score and training efficiency, we design a *structured word2vec* embedding method. We tokenize the pathnames in each container image layer by adapting Columbus [50] used in Praxi's *tagset* generation [12]. Other alternative encoding methods [13], [36], [51] are also applicable. Then, we build a *token-frequency table* (Table I) for tokens and package labels. Each row represents a layer, where each entry shows the count of a token or the *one-hot-encoding* of a label.

Our *structured* word2vec ML pipeline includes many word2vec encoders focusing on distinct sets of similar packages and installation fingerprints. Package *similarity* is defined as cosine similarity using *token-frequency* vectors of different packages in the trainset. Then, as in Fig. 4a, we cluster similar packages above a similarity threshold and assign one or more clusters to each encoder (*Submodel Routing*). Next, we generate a feature embedding dictionary for each word2vec encoder. This method allows adding packages to existing or new clusters incrementally. Furthermore, different assignments of clusters, i.e. combining clusters to one encoder, can balance submodel training / inference budget and precision / F1-score target.

Clustering similar packages is necessary to enable high precision and F1-score for our structured model, where we pair each encoder with a submodel to make predictions (Sec. III-A3). However, duplicate tokens across various fingerprints could lead to false positives from different submodels, unable to distinguish unique packages from duplicate tokens, e.g., "vine==5.0.0" vs. "vine==1.3.0" mentioned in the introduction. We detail this challenge in Sec. III-B.

3) Structured Model based on Data Distribution: We build a structured model comprising multiple XGBoost [52] submodels to predict installed packages. We chose XGBoost for its high F1-score and explainability for our task. As shown in Fig. 4a, for each word2vec encoder, we train an XGBoost submodel for *similar* package labels, using corresponding tokens and samples. During inference, as shown in Fig. 4b, sample tokens are processed by a *subset* of encoders, with each encoder transforming recognized tokens into feature embeddings within its output space. Corresponding XGBoost submodels then infer package labels based on the embeddings. Next, we union outputs from all submodels as the final predictions.

Our structured model also enables invoking a subset of submodels during inference and saves resource costs. For example, considering "fiona==1.9.5" and "vine==5.0.0" with low cosine similarity, the same tokens are from common Python package file pathnames, e.g., "cache", "METADATA",


Sub-Model A
Sub-Model B
Word2Vec Encoder A
Word2Vec Encoder B
Package Cluster 1
Package Cluster 2
Package Cluster 3
Cosine Similarity
Create a new cluster if similarity is less than τ
Sample 1 Token1 Count Token2 Count
Sample 2 Token1 Count Token2 Count
L2 L3 True Label
Confidence 0.96 0.97 L2,L3
Labels L2 Confidence 0.96
Union
Labels L1 L3 Confidence Low 0.97
Mapping Table A
Mapping Table B
Token1: Index1
Token2: Index2
Token3: Index1
Fingerprint 1 Token1 Token2 Token3

(a) Submodel Routing (b) Inference pipeline

Fig. 4: Fig. 4a illustrates how we cluster package labels with pairwise cosine similarity of package tokens and then assign clusters of labels to submodels. In Fig. 4b, each encoder maps its known tokens from a sample to feature embeddings, and the corresponding XGBoost submodel makes inferences.

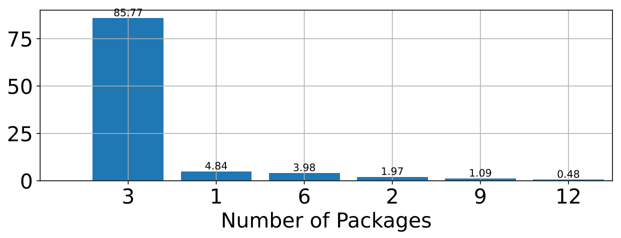


Fig. 5: % of duplicate tokens present in different numbers of package file changes: We focus on the top 6 most duplicated tokens. Most tokens duplicate in three packages, as we include three versions of each package in the dataset. Only less than 5% of tokens can uniquely identify some packages.

and "WHEEL" which the classifier would likely assign zero importance during training. Thus, with the encoder mapping table, we can identify tokens with corresponding submodel features with non-zero importance. During inference, we check if any important tokens are present in the installation fingerprint and invoke the corresponding word2vec encoder and submodel.

By focusing on similar packages, each submodel pays more attention to distinctive features and minimizes tree-splitting operations. The following section details the model decomposition, similarity analysis, and performance.

B. Structured Model Motivated by Data Distribution

This section analyzes the training time efficiency and high F1-score of structured XGBoost models given XGBoost internals and installation fingerprint distribution. Then, we formulate an optimization problem balancing training time and F1-score and use an iterative solution to configure our structured model.

1) Training Times: The tree split finding operations in the XGBoost greedy exact search [52][2] iterate through each input

[2]Note that the alternative approximate greedy algorithm focusing on bucketing samples to reduce split finding operations in XGBoost [52] is orthogonal to our discussion.

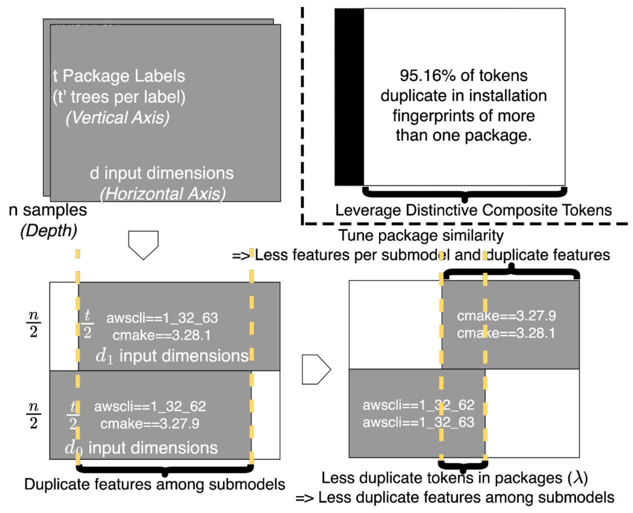


Fig. 6: The volume of grey blocks indicates the splitting operation to train XGBoost models in the structured model, given the number of labels (vertical axis), input dimensions (horizontal axis), and number of samples (depth axis). The upper left single grey block represents one model, and each of the lower figures includes two submodels. From lower left to right, we tune τ and lower λ to minimize duplicate features between submodels, which minimizes false positives.

feature (d) of each sample (n) of each label (t) to identify the optimal split [52]. Thus, given constant $t^{'}$ trees per label and boosting trees with height 1, splitting operations ($Ops^{(0)}$), as shown by the volume of grey brick in the top left of Fig. 6, is

$$Ops^{(0)} = (n \; log \; n + n)d \; t \; t^{'}$$

Then, decomposing the model provides an exponential drop in training time. For example, halving the number of labels per submodel (t) also halves samples to train each submodel (n) and reduces the input dimensions of each submodel, which drops the split-finding operations of all submodels quadratically.

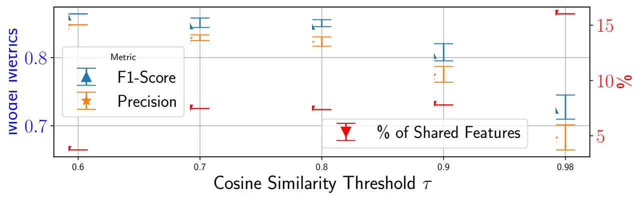


Fig. 7: Prediction metric (left y-axis) and duplicate features in more than one submodel (right y-axis) vs Cosine Similarity threshold to cluster package labels. Setting τ too high leads to too few labels in a cluster, which would lead to more duplcate features among submodels. While setting τ too low, assigning all labels into one cluster, leads to high prediction metrics but also long incremental training time for the large clusters.

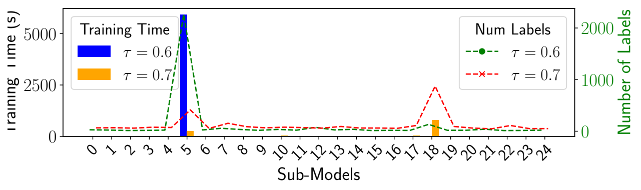


Fig. 8: Training time of each submodel and the number of package labels assigned to submodels in an illustrative example. With higher τ, some clusters include the majority of package labels, which leads to long training time for certain submodels and diminishes the benefit of model decomposition.

In the lower left of Fig. 6, the total operations (Ops) becomes

$$Ops = (\frac{n}{2}\ log\ \frac{n}{2} + \frac{n}{2})d_0\ \frac{t}{2}\ t^{'} + (\frac{n}{2}\ log\ \frac{n}{2} + \frac{n}{2})d_1\ \frac{t}{2}\ t^{'},$$

where d_0 and d_1 are input dimensions for each submodel which depend on the number of duplicate features. Thus,

$$\frac{1}{4}(n\ log\ \frac{n}{2} + n)d\ t\ t^{'} \leq Ops < \frac{1}{2}Ops^{(0)}$$
$$d \leq d_0 + d_1 \leq 2d, d_0 > 0, d_1 > 0.$$

2) F1 Scores: We now analyze whether the structured model design can preserve accuracy compared to the traditional monolithic model design used in DeltaSherlock [11]. For our training dataset in Fig. 5, about 95.16% ($100\% - 4.84\%$) of tokens are present in installation fingerprints of more than one package. This distribution implies if we randomly assign packages to submodels, there will be many duplicate input features among submodels (lower left of Fig. 6). During inference, if any of the features are assigned a higher weight for prediction, the structured model would suffer from false positive software discoveries and low precision. To evaluate the relationship between precision and the number of duplicate features, we train a structured model using a training dataset of $1,000$ packages, with 3 versions and 24 fingerprints per version. Then we test the model performance using a test dataset incorporating $47,274$ combinations of 2 different packages with varied versions. As shown in Fig. 7, with 16% of the features duplicated in more than one submodel, the structured model's precision drops to about 0.68, but with less than 5% duplicate features, the precision improves to 0.85. With more features duplicated in more than one submodel, the structured model tends to have worse prediction results. Next, we introduce our remedies to preserve model precision.

3) Structured Model Improvements: As shown in the lower right of Fig. 6, we propose two methods to minimize duplicate features. *Submodel Routing* optimizes package similarity (based on cosine similarity of package installation fingerprints or considering package versions as a similarity heuristic) and filter duplicate tokens among package installation fingerprints.

Cosine Similarity based *Submodel Routing*. Assigning similar packages to different submodels results in more duplicate features, negatively affecting structured model precision. Thus, following Sec. III-A2, we design a package *similarity* threshold ($\tau \in [0, 1]$) to cluster *similar* packages and distribute these clusters randomly across submodels. This approach allows incrementally adding new packages to existing clusters and retraining the particular submodel or forming new clusters and submodels. However, tuning τ is non-trivial. We train a structured model with $1,000$ packages with 3 versions per package and 24 installation fingerprints per package/version. Then, we evaluate the model performance given different τ with a test dataset, including $47,274$ combinations of 2 packages with different versions. A high τ creates too many clusters and causes low precision caused by many duplicate features after random assignment to submodels ($\tau = 0.9$ in Fig. 7). A low τ clusters most packages into a few clusters, which diminishes the training time benefit of the structured model compared to existing work with the monolithic model. In Fig. 8, setting $\tau = 0.6$ assigns the majority of the labels ($2,265$ out of $3,000$) to one submodel and the training time for that submodel is $5,924.64s$ ($2,544.9$ times that of the average of other submodels). This skewness issue causes the structured model to suffer from frequently retraining the large submodel when introducing new labels.

Package Version based *Submodel Routing*. We relax τ given the longest submodel training time satisfies a service level objective. Such tuning steps require extensive experimentation, which wastes computation resources. Instead, we propose clustering versions of each package as a heuristic solution, which balances the number of clusters and cluster sizes as a regularizer. This clustering method works well, as shown in our evaluation (Sec. IV). Versions with minor changes tend to have high cosine similarity. Versions of one package with significant changes still tend to have low cosine similarity with other packages, so false positives remain minimized.

Filter for Duplicate Tokens. We apply feature selection to tokens during structured model initialization and new submodel additions to minimize duplicate features across submodels, satisfying a service level objective of prediction precision (A_{SLO}). We decrease a threshold ($\lambda \in [0, \|Packages\|]$) for the maximum number of occurrences of duplicate tokens among observed packages to reduce false positives for previously trained package labels. However, lower λ removes more tokens, which can cause low recalls, i.e., the model always has low confidence for some labels. Also, we might eventually need to retrain the existing submodels with filtered package fingerprints for optimal precision. Thus, tuning this feature selection tool is non-trivial, and it improves the most precision with minimal training overhead when initializing the structured model.

Correctly optimizing τ and λ minimizes duplicate features between submodels, which minimizes split-finding operations and maintains high F1-score. Next, we propose our hyper-parameter tuning method for our structured model.

4) Configuring Structured Models: We construct a multi-objective optimization formulation to minimize training time per submodel ($T(.)$) while maximizing structured model precision ($A(.)$) in Eq. 1.

A	Weighted Precision across labels
R	Weighted Recall across labels
T	Training time per submodel
T_{worst}	Slowest training time among submodels
n_{dat}	Total number of samples in trainset
T^0,t^0,d^0,n^0,A^0,R^0	Last observed/simulated values
τ	Threshold for pairwise cosine similarity
λ	Threshold for tokens across packages
$L, tokens$	Clusters of labels and corresponding tokens

TABLE II: Symbols in Alg. 1

$$\begin{aligned} &\min_{t,d,n}(T(t,d,n) - A(t,d,n)) \\ &s.t.\ A(C_{dup\_token}(\tau, \lambda, t, d)) \geq A_{SLO} \\ &T(t,d,n) = T^0 \frac{(n\ log\ n + n)d\ t}{(n^0\ log\ n^0 + n^0)d^0\ t^0} \end{aligned} \tag{1}$$

We optimize for a "sweet spot" for both training time per submodel and model precision. For XGBoost submodels, training time depends on split-finding operation counts given the parameters d, t, and n. We can estimate training time (T) with corresponding d, t and n, given training time observation (T^0) and corresponding d^0, t^0 and n^0. Meanwhile, the weighted precision of all packages drops when the duplicate features among submodels ($C_{dup\_token}(.)$) increases, which yields extra false positives. Thus, we bound model precision with a Service Level Objective (A_{SLO}). We can minimize the duplicate features by tuning the threshold of pairwise cosine similarity (τ) for tokens of packages when forming clusters and filter tokens present in installation fingerprints of multiple packages (λ). We detail the optimization steps with Alg. 1 and Table. II.

The first loop tunes submodel size by additively decreasing t to meet the incremental training time Service Level Objective (T_{SLO}) with estimation (line 6). We define two profile functions $prof_d(t)$ and $prof_n(t)$ that return the largest dimension and sample size for any t packages from the trainset. The second loop evaluates the precision of the structured model by clustering labels and corresponding tokens ($L, tokens$) given $\tau \in [0,1]$ or heuristically clustering versions of each package. We evenly assign clusters to submodels. If we still violate A_{SLO}, we perform the third loop, which decreases λ to filter out more duplicate tokens. This loop halts when we observe the weighted recall for all labels (R) drops.

IV. Evaluation

We assess the scalability of our structured model by comparing training time and F1-score against existing systems using Chameleon Cloud [53] nodes. We also present the inference time of our *PraxiPaaS* NERC deployment. The evaluation uses Package Version based *Submodel Routing*. We evaluate against DeltaSherlock [11] and Praxi [12], which use the monolithic word2vec ML pipeline. DeltaSherlock is implemented as a special case of the structured model by training a single submodel for all package labels. For Praxi, we use their open-source implementation with Vowpal Wabbit and proactive classifier initialization and training to study the model calibration costs.

Algorithm 1 Hyper-Parameter Tuning for Submodels Init

```
Require: $T^0$, $t^0$, $d^0$, $n^0$, $prof_d$, $prof_n$, $T_{SLO}$, $A_{SLO}$
Ensure: $t$, $d$, $n$, $\tau_{best}$, $\lambda_{best}$
 1: $\lambda_{best} = \lambda = \|packages\|$
 2: $\tau_{best} = \tau = 1$
 3: $T, t = T^0, t^0$
 4: while $T \geq T_{SLO}$ do
 5:     $t = t - 1$
 6:     $T = T^0 \frac{(prof_n(t)\log(prof_n(t)) + prof_n(t))prof_d(t,\lambda)\ t}{(n^0 \log n^0 + n^0)d^0\ t^0}$
 7: end while
 8: $A, A^0, T_{worst} = 0, -1, T$
 9: while $A \leq A_{SLO}$ and $A^0 < A$ and $T_{worst} \leq T_{SLO}$ and
    $\tau \geq 0$ do
10:     $\tau_{best}, A^0 = \tau, A$
11:     $L, tokens = clustering(tokens\_per\_package, \tau)$
12:     $\tau = \tau - 0.1$
13:     $A, T_{worst} = eval(t, prof_d(t, \lambda_{best}), prof_n(t), L, tokens)$
14: end while
15: $L, tokens = clustering(tokens\_per\_package, \tau_{best})$
16: $R, R^0, A = 0, -1, A^0$
17: while $R^0 \leq R$ and $A \leq A_{SLO}$ do
18:     $\lambda_{best}, R^0 = \lambda, R$
19:     $\lambda = \lambda - 1$
20:     $A, R = eval(t, prof_d(t, \lambda_{best}), prof_n(t), L, tokens)$
21: end while
```

A. System Setup & Implementation

We provision Chameleon Cloud nodes with 128 cores and 256 GB memory for the model evaluations. For our NERC deployed *PraxiPaaS* (Fig. 3), we configure each component to have 4 cores and $4GB$ of memory in the NERC Openshift AI Pipeline. To evaluate inference latency, we sample images with 7 layers each and test with different image batch sizes. We configure 32 threads for data loading and XGBoost model training for the best training speed. And we use 100 boosting trees per label to preserve F1-score to extensively evaluate non-linear feature correlations, as we need more trees to fit the potential composite features (illustrated in Fig. 5). Other parameters are shown in our repo [54].

B. Datasets

We construct a list of the most frequently downloaded Python packages from PyPI [14] and GitHub trending Python repos [55]. The list includes $3,000$ labels, i.e., $1,000$ packages and each with 3 different versions.

We collect the installation file changes by building container images and read the file changes by layers. The single-label dataset has 24 fingerprints per label by reading the file changes when installing a version of a package in a unique base container image with all dependencies installed. Similarly, the multi-label dataset includes installation file changes of choosing any 2 labels from the $3,000$ labels, filtering the combinations of

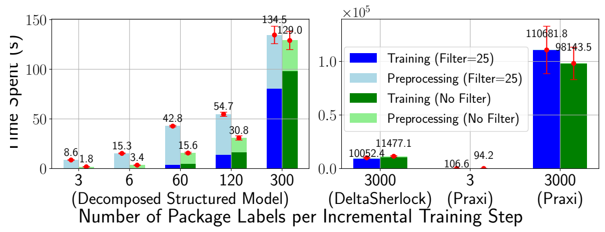


Fig. 9: Training and data preprocessing times (y-axis) versus numbers of packages per submodel (x-axis), DeltaSherlock (single submodel for all labels), and Praxi adding 3 packages or retraining the complete model. Preprocessing includes word2vec encoder building (for structured model and DeltaSherlock) and duplicate token filtering (for all methods). Training of *PraxiPaaS* is more scalable than DeltaSherlock or Praxi.

the same packages but different versions. We then sub-sample to $47,274$ installation file changes.

We perform a 4-fold cross-validation for single-label fingerprints, and we also randomize the package clusters attributed to each submodel in the structured model 10 times to evaluate the stability of training and inference. Next, we test the trained models in the previous steps with the multi-label fingerprints. We show the 95% confidence intervals in our plots.

C. Scalability Analysis: Training Time

We compare the incremental training times of our structured model, given different numbers of labels per submodel, and perform 4-fold cross-validation over the single-label dataset.

We evaluate Package Version based *Submodel Routing* and filter tokens with certain occurrences in the dataset. As Sec. III-B3 discusses, versions of the same package tend to have high pairwise cosine similarity, which enhances training time and model precision by minimizing features per submodel and duplicate features across submodels. Meanwhile, those versions with low cosine similarity tend to have a few duplicate features compared with other packages. So assigning it to the same cluster with other versions of the same package saves computation resources for finding similar packages and minimizes duplicate features across submodels.

Fig. 9 shows data preprocessing time for word2vec encoder building (for Structured Model and DeltaSherlock) and duplicate token filtering (for all methods) and training time given different numbers of labels per incremental training step. Our dataset contains 3 versions of each package, so we selectively present the training time given 1, 2, 20, 60, and 100 cluster(s) per submodel (i.e., 3, 6, 60, 120 and 300 package labels per submodel). We implement DeltaSherlock as a special case in the structured model and measure the training time of one submodel trained for all $3,000$ labels. Praxi uses OAA reduction in Vowpal Wabbit and proactively builds $3,000$ classifiers for all $3,000$ package labels with the open-sourced Praxi. We show incremental training time for 3 packages when trained labels are less than the proactively built classifiers. Then, we also show the complete model retraining time with all $3,000$ labels.

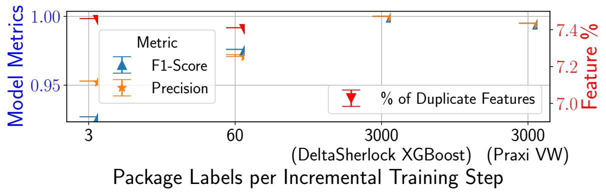


Fig. 10: The weighted prediction quality metrics (left y-axis) and % of duplicate features among submodels (right y-axis) by number of labels per submodel: By clustering the different versions of each packages, our structured model can maintain high precision compared to DeltaSherlock (XGBoost) and Praxi (Vowpal Wabbit with proactive classifier building).

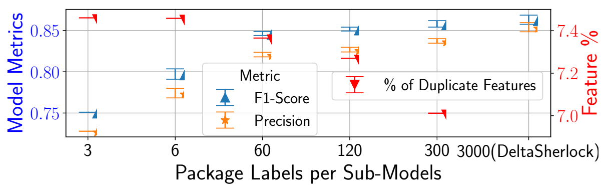


Fig. 11: The weighted prediction quality metrics (left y-axis) and duplicate features % (right y-axis) by number of labels per submodel: Overall, by clustering the different versions of each package, our structured model can maintain high precision compared to DeltaSherlock.

The incremental training time drops quadratically as the number of labels per submodel reduces. Preprocessing time can exceed training time when there are only a few labels per submodel, given parallelization implementation and data size. Overall, our structured model design is more scalable than monolithic models (Deltasherlock with XGBoost and Praxi with proactive classifier building in VW) for incremental training steps.

D. Scalability Analysis: Single-Label Prediction

We study the software discovery prediction with 4-fold cross-validation over our single-label dataset. We include our Structured Model with 1 or 20 package cluster(s) per submodel (i.e., 3 or 60 package labels per model), as well as DeltaSherlock and Praxi.

In Fig. 10, single-label predictions achieve above 0.925 F1-score for all methods. While considering the training time presented in the previous section, the structured model is still the most scalable, compared to Deltasherlock with XGBoost and Praxi using proactive classifier building.

E. Scalability Analysis: Multi-Label Prediction

In this section, we evaluate the prediction quality of our *Structured Model* with 1 or 20 package cluster(s) per submodel (i.e., 3 or 60 package labels per model) and monolithic models (DeltaSherlock) using our multi-label dataset, where each installation fingerprint contains 2 package installations. We use our models trained during cross-validation using the single-label trainset. However, Praxi is designed to train with a multi-label

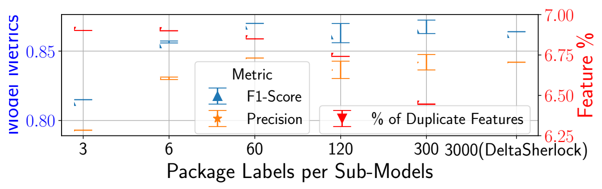


Fig. 12: The weighted prediction quality metrics (left y-axis) and duplicate features % (right y-axis) by number of labels per submodel and filter tokens present in file changes of more than 25 packages: We see similar behavior as in Fig. 11, but overall higher precision with filter.

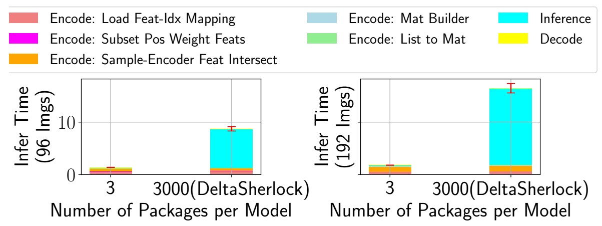


Fig. 13: Structured versus Monolithic Model inference time in NERC includes word2vec encoding, prediction generating, and word2vec decoding. We show the *model* inference times of 96 or 192 container images in a batch.

dataset first and then tested with multi-label data [12], which is not scalable since the number of package combinations exponentially increases when adding new packages to the model. Thus, we do not show Praxi results to save space.

As shown in Fig. 11, the multi-label prediction quality of our structured model is maintained similarly to our implementation of DeltaSherlock using the XGBoost model. And after filtering duplicate tokens that are present in file changes of more than 25 packages, structured model precision can be close to DeltaSherlock (Fig. 12), but we potentially tradeoff recall (Sec. III-B3). Notice that recalls are about 0.98 for all our evaluations. These results show that we can achieve a good balance between fast training time and high F1-score by tuning package labels per submodel, clustering versions of each package, and filtering duplicate tokens.

F. NERC Deployment Inference Latency Tradeoffs

This section shows the inference latencies of *PraxiPaaS* (Fig. 3) using the structured model versus the monolithic model (our implementation of DeltaSherlock using XGBoost). The end-to-end latency of our ML pipeline includes identifying running containers using cluster observability, pulling images, package installation fingerprint generation, tokenization, word2vec encoding (Encode), making predictions (Inference), and decoding the predictions into package names with versions (Decode). We focus on the last three steps as others are the same in structured and monolithic models.

Our structured ML pipeline only encodes and infers fingerprints with positive feature importances according to submodels. Fig. 13 presents the stages of model inference time of 96 or 192 container images in a batch using the structured model with 3 packages per submodel or a monolithic model trained for all $3,000$ labels. For Encode step, we highlight substeps including #1 Loading feature to matrix column index mapping for each word2vec encoder (Load Feat-Idx Mapping), #2 Identifying positive feature importance (Subset Pos Weight Feats), #3 Finding intersection between fingerprint tokens and encoder features (Sample-Encoder Feat Intersect), #4 Vectorizing tokens (Mat Builder), and 5) Converting vectors to a matrix for inference (List to Mat). Substep #3 minimizes tokens for vectorization, reducing inference times of structured and monolithic models. Meanwhile, since each pair of encoder and submodel focuses on a small subset of labels and corresponding tokens, this step also determines the minimal subset of submodels to make inferences. However, a monolithic model requires vectorizing all input tokens. Thus, the structured model saves model inference time over monolithic model designs (Fig. 13: 0.027s (0.032s) for *PraxiPaaS* and 7.45s (14.63s) for DeltaSherlock with container image batch size 96 (192).)

G. Discussion

Overall, *PraxiPaaS* achieves fast incremental training and high F1-score with a decomposable structured model design. The empirical hyper-parameter tuning in Alg. 1 can be costly. However, the package version-based submodel routing method provides a resource-efficient approximation for package similarity measurements. Other parameters, including the labels per model, features, and samples, can be determined given a tolerance of incremental training time. We will explore an accurate and cheap model performance estimator in our future work. Also, model and resource provisioning for submodels introduce challenges to achieving low latency and cost efficiency. Existing methods [56], [57] can be applied to submodels serving.

V. Conclusion

We introduce *PraxiPaaS*, a scalable and cost-efficient container package discovery framework in the PaaS cluster using a decomposable ML method for SBOM generation and vulnerable package detection. Recent ML frameworks for package discovery have limited scalability in PaaS settings.

We evaluate the scalability tradeoffs of our structured word2vec ML pipeline utilizing the data distribution of package installation fingerprints in terms of incremental training time, F1-score, and inference time. To configure our structured model, we introduce an optimization formulation with an algorithm to address it. Our results show that structured ML pipeline achieves quick incremental training, high F1-score, and low inference time compared to monolithic models. Our next steps include evaluating other word2vec encoders, building an F1-score estimator to save resource costs for structured model configuration tuning, and studying model and resource provisioning methods to maintain low latency and cost-efficiency.

REFERENCES

[1] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, Yoday, and Tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[2] Redhat, "Red Hat Research Report: Cloud-native development outlook," 2021. [Online]. Available: https://www.redhat.com/en/resources/cloud-native-development-outlook-whitepaper

[3] S. Carmody, A. Coravos, G. Fahs, A. Hatch, J. Medina, B. Woods, and J. Corman, "Building resilient medical technology supply chains with a software bill of materials," *npj Digital Medicine*, vol. 4, no. 1, p. 34, Feb 2021. [Online]. Available: https://doi.org/10.1038/s41746-021-00403-w

[4] J. T. Stoddard, M. A. Cutshaw, T. Williams, A. Friedman, and J. Murphy, "Software Bill of Materials (SBOM) Sharing Lifecycle Report," Idaho National Laboratory (INL), Idaho Falls, ID (United States), Tech. Rep., 4 2023. [Online]. Available: https://www.osti.gov/biblio/1969133

[5] NERC, "NERC Technical Documentation," 2023. [Online]. Available: https://nerc-project.github.io/nerc-docs/

[6] Google, "Announcing OSV-Scanner: Vulnerability Scanner for Open Source," 2023. [Online]. Available: https://security.googleblog.com/2022/12/announcing-osv-scanner-vulnerability.html

[7] D. Stenberg, "Severity HIGH Security Problem to Be Announced with Curl 8.4.0 on Oct 11," 2023. [Online]. Available: https://github.com/curl/curl/discussions/12026

[8] Google, "OSV-Scanner," https://github.com/google/osv-scanner, 2023.

[9] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1509–1526. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.10179304

[10] H. Chen, S. S. Duri, V. Bala, N. T. Bila, C. Isci, and A. K. Coskun, "Detecting and Identifying System Changes in the Cloud via Discovery by Example," in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 90–99.

[11] A. Turk, H. Chen, A. Byrne, J. Knollmeyer, S. S. Duri, C. Isci, and A. K. Coskun, "Deltasherlock: Identifying Changes in the Cloud," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 763–772.

[12] A. Byrne, E. Ates, A. Turk, V. Pchelin, S. Duri, S. Nadgowda, C. Isci, and A. K. Coskun, "Praxi: Cloud Software Discovery That Learns From Practice," *IEEE Transactions on Cloud Computing*, vol. 10, no. 2, pp. 872–884, 2022.

[13] W. S. El-Kassas, C. R. Salama, A. A. Rafea, and H. K. Mohamed, "Automatic Text Summarization: A Comprehensive Survey," *Expert Systems with Applications*, vol. 165, p. 113679, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417420305030

[14] H. van Kemenade, M. Thoma, R. Si, and Z. Dollenstein, "hugovk/top-pypi-packages: Release 2023.10," Oct. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8396367

[15] Redhat, "Cloud-Native CI/CD with OpenShift Pipelines," 2019. [Online]. Available: https://www.redhat.com/en/blog/cloud-native-ci-cd-with-openshift-pipelines?extIdCarryOver=true&sc_cid=701f2000001OH7JAAW

[16] Z. Qin, V. Petricek, N. Karampatziakis, L. Li, and J. Langford, "Efficient online bootstrapping for large scale learning," *arXiv preprint arXiv:1312.5021*, 2013.

[17] A. L. Suárez-Cetrulo, D. Quintana, and A. Cervantes, "A Survey on Machine Learning for Recurring Concept Drifting Data Streams," *Expert Systems with Applications*, vol. 213, p. 118934, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417422019522

[18] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=B1ckMDqlg

[19] S. Gross, M. Ranzato, and A. Szlam, "Hard Mixtures of Experts for Large Scale Weakly Supervised Vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6865–6873.

[20] Redhat, "Red Hat OpenShift Data Science," 2023. [Online]. Available: https://www.redhat.com/en/technologies/cloud-computing/openshift/openshift-data-science

[21] ——, "Red Hat Advanced Cluster Management for Kubernetes," 2023. [Online]. Available: https://www.redhat.com/en/technologies/management/advanced-cluster-management

[22] Microsoft, "SBOM-Tool," 2024. [Online]. Available: https://github.com/microsoft/sbom-tool

[23] L. Foundation, "System Package Data Exchange (SPDX)," 2023. [Online]. Available: https://spdx.dev/

[24] IBM, "Agentless System Crawler," https://github.com/cloudviz/agentless-system-crawler, 2018.

[25] A. Zerouali, V. Cosentino, G. Robles, J. M. Gonzalez-Barahona, and T. Mens, "ConPan: A Tool to Analyze Packages in Software Containers," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 592–596.

[26] Anchore, "Syft," https://github.com/anchore/syft, 2023.

[27] ——, "Grype," https://github.com/anchore/grype, 2023.

[28] Redhat, "Claircore," https://github.com/quay/claircore, 2023.

[29] R. Elwell and R. Polikar, "Incremental Learning of Concept Drift in Nonstationary Environments," *IEEE Transactions on Neural Networks*, vol. 22, no. 10, pp. 1517–1531, 2011.

[30] G. Ditzler, "A Study of An Incremental Spectral Meta-Learner for Nonstationary Environments," in *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, pp. 38–44.

[31] Anchore, "Anchore," 2023. [Online]. Available: https://anchore.com/

[32] Redhat, "Clair," https://github.com/quay/clair, 2023.

[33] A. Mohan, S. Nadgowda, B. Pipaliya, S. Varma, S. Suneja, C. Isci, G. Cooperman, P. Desnoyers, O. Krieger, and A. Turk, "Towards Non-Intrusive Software Introspection and Beyond," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2020, pp. 173–184.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 307–320.

[35] A. Mohan, A. Turk, R. S. Gudimetla, S. Tikale, J. Hennesey, U. Kaynar, G. Cooperman, P. Desnoyers, and O. Krieger, "M2: Malleable Metal as a Service," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 61–71.

[36] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," *arXiv preprint arXiv:1301.3781*, 2013.

[37] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge Distillation: A Survey," *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.

[38] J. Shao, Y. Mao, and J. Zhang, "Learning Task-Oriented Communication for Edge Inference: An Information Bottleneck Approach," *IEEE J.Sel. A. Commun.*, vol. 40, no. 1, p. 197–211, jan 2022. [Online]. Available: https://doi.org/10.1109/JSAC.2021.3126087

[39] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, and T. Abdelzaher, "Deep Compressive Offloading: Speeding up Neural Network Inference by Trading Edge Computation for Network Latency," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, ser. SenSys '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 476–488. [Online]. Available: https://doi.org/10.1145/3384419.3430898

[40] Y. Matsubara and M. Levorato, "Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks," in *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 2272–2279.

[41] T. Liang, J. Glossner, L. Wang, S. Shi, and X. Zhang, "Pruning and Quantization for Deep Neural Network Acceleration: A Survey," *Neurocomputing*, vol. 461, pp. 370–403, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221010894

[42] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[43] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun, "MetaPruning: Meta Learning for Automatic Neural Network Channel Pruning," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 3296–3305.

[44] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "SplitNet: Learning to Semantically Split Deep Networks for Parameter Reduction and Model Parallelization," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1866–1874.

[45] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently Scaling Transformer Inference," *Proceedings of Machine Learning and Systems*, vol. 5, 2023.

[46] K. Huang and W. Gao, "Real-Time Neural Network Inference on Extremely Weak Devices: Agile Offloading with Explainable AI," in *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, ser. MobiCom '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 200–213. [Online]. Available: https://doi.org/10.1145/3495243.3560551

[47] S. S. Ogden, X. Kong, and T. Guo, "PieSlicer: Dynamically Improving Response Time for Cloud-Based CNN Inference," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 249–256. [Online]. Available: https://doi.org/10.1145/3427921.3450256

[48] D. Freitag, "Machine Learning for Information Extraction in Informal Domains," *Machine Learning*, vol. 39, no. 2, pp. 169–202, May 2000. [Online]. Available: https://doi.org/10.1023/A:1007601113994

[49] Docker, "Images and layers," https://docs.docker.com/storage/storagedriver/#images-and-layers, 2023.

[50] S. Nadgowda, S. Duri, C. Isci, and V. Mann, "Columbus: Filesystem Tree Introspection for Software Discovery," in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 67–74.

[51] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50, http://is.muni.cz/publication/884893/en.

[52] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.

[53] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons Learned from the Chameleon Testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[54] "PraxiPaas," https://github.com/ai4cloudops/Praxi-Pipeline, 2024.

[55] Github, "Github Trending," 2023. [Online]. Available: https://github.com/trending/python?since=monthly

[56] Z. Hong, J. Lin, S. Guo, S. Luo, W. Chen, R. Wattenhofer, and Y. Yu, "Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1039–1053. [Online]. Available: https://doi.org/10.1145/3627703.3629567

[57] A. Raza, Z. Zhang, N. Akhtar, V. Isahagian, and I. Matta, "LIBRA: An Economical Hybrid Approach for Cloud Applications with Strict SLAs," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*, 2021, pp. 136–146.