LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-857729

Learning to Predict and Improve Build Successes in Package Ecosystems

H. Menon, D. Nichols, A. Bhatele, T. Gamblin

November 29, 2023

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Learning to Predict and Improve Build Successes in Package Ecosystems

Harshitha Menon*
Lawrence Livermore National Laboratory
Livermore, California, USA
harshitha@llnl.gov

Daniel Nichols*
University of Maryland
College Park, Maryland, USA
dnicho@umd.edu

Abhinav Bhatele
University of Maryland
College Park, Maryland, USA
bhatele@cs.umd.edu

Todd Gamblin
Lawrence Livermore National Laboratory
Livermore, California, USA
tgamblin@llnl.gov

ABSTRACT

Software has become increasingly complex, with a typical application depending on tens or hundreds of packages. Finding compatible versions and build configurations of these packages is challenging. This paper presents a method to learn the likelihood of software build success, and techniques for leveraging this information to guide dependency solvers to better software configurations. We leverage the heavily *parameterized* package recipes from the Spack package manager to produce a training data set of builds, and we use Graph Neural Networks to learn whether a given package configuration will build successfully or not. We apply our tool to the U.S. Exascale Computing Project's software stack. We demonstrate its effectiveness in predicting whether a given package will build successfully. We show that our technique can be used to improve the solutions generated by dependency solvers, reducing the need for developers to find working builds by trial and error.

1 INTRODUCTION

Modern software has become incredibly complex. Developers frequently reuse software components, such as libraries, frameworks, and APIs, to expedite the development process. Reusing software saves time and separates concerns, allowing developers to rely on well-established implementations without rewriting them. While reusable software components enable rapid development and feature-rich applications, they introduce complexity to software management. Different components may have different version and configuration requirements on their dependencies, leading to compatibility issues requiring additional efforts for integration. Developers must constantly expend effort to keep pace with rapidly changing software dependencies.

In order to tackle this complexity, modern software ecosystems rely on automated package managers like APT, Cargo, Maven, NPM, and Spack. These tools analyze compatibility constraints among different packages and select a compatible set of package versions to install. The process of selecting a *consistent* version configuration is known to be NP-complete [7, 26, 28, 1, 41], and most package managers use sophisticated solvers to find valid package configurations that satisfy compatibility constraints. However, these constraints are specified by package maintainers, and they are inherently imperfect because maintainers cannot test all package combinations for compatibility. In practice, a *valid* solution may still fail to build, if the constraints it is based on are not sound.

Package management systems adopt different strategies when selecting package versions, for instance, always selecting the most recent version that satisfies an open ended range. Any such version selection mechanism can produce errors as long as it relies on imprecise, hand-annotated version constraints. As the number of dependencies grows, the likelihood of an incompatibility increases, particularly for large, transitive dependency stacks. The software ecosystem is too large and there are too many exceptions and corner cases to come up with a concise set of rules for all package recipes. Package maintainers cannot test every version combination, particularly among deep, transitive dependencies they may not even be aware of. Therefore, the practice of user-defined package version constraints is inherently susceptible to errors.

Complexities arising from such errors is particularly common in the scientific computing, High-Performance Computing (HPC), and ML ecosystems, where it is common to port software to new architectures and platforms. Preexisting builds, such as those provided by a standard Linux package manager, don't necessarily work on these hardwares. Existing dependency constraints are not sufficient as these ecosystems are dealing with new hardware, exotic compilers and libraries, where compatibility information is not known yet. Package maintainers typically specify open-ended compatibility ranges to allow new versions to be integrated easily, but the correct ranges for one platform may be slightly different from those for another, and even very slight differences can lead to build errors [15, 25, 21, 10]. Finding a working configuration and updating constraints can be a very expensive trial-and-error exercise that can take days or weeks to converge to a working build. Moreover, finding one working build does not necessarily tell us how best to update compatible version ranges.

Our aim in this paper is to understand build incompatibilities, predict bad configurations, and assist developers in managing version constraints by automatically generating constraints based on empirical information. With recent advances in machine learning we can learn from a relatively small set of examples and predict what options will most likely lead to success. Graph neural networks are uniquely suited to this problem as they can model and

*These authors contributed equally to this work.

Conference'17, July 2017, Washington, DC, USA
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

learn node properties and edge relationships in general graphs. This, in conjunction with the ability of neural networks to find patterns and model extremely complex systems, is uniquely suited to modeling complex software dependency graphs.

In this paper, we leverage cutting-edge AI technology and advanced package management methodologies to address the challenges of managing software ecosystems. We use graph neural networks (GNNs) to analyze a prominent software ecosystem in HPC, the Exascale Computing Project (ECP) software stack E4S. By using the ECP's E4S stack as an example, and leveraging Spack's parameterized package recipes, we demonstrate that GNNs can be effectively trained to understand the build incompatibilities in a large software ecosystem and identify configurations that will not work, *without* the need to actually build them. Moreover, we use the compatibility information extracted from the GNN model and integrate it into Spack package manager's version selection mechanism to steer the package solver towards likely-to-build solutions.

Our main contributions are the following:

- *BuildCheck*, a Graph Neural Network based approach that predicts the outcome of builds for different package configurations.
- Apply transfer learning, which involves transferring the learning from a low-cost source data to a target data, and use it when limited resources are available for collecting data at scale for a target application.
- A detailed experimental evaluation of our approach over 45, 837 builds from the E4S scientific software ecosystem.
- A methodology for using the outputs of *BuildCheck* to improve build likelihood in Spack.

Furthermore, we seek to answer the following research questions in our work.

RQ1 *Can* BuildCheck *predict the build outcome of various package configurations with high accuracy?* We demonstrate that, after training, *BuildCheck* is able to correctly classify build outcomes for 91% of the builds in the testing dataset.

RQ2 *Can self-supervised pre-training be used to reduce the need of expensive build data to train the model?* When trained with pre-training + fine-tuning, *BuildCheck* is shown to achieve comparable or better performance to the base model. In particular, it gets up to 3% improvement for low numbers of training samples.

RQ3 *Can the outputs of* BuildCheck *be used to select package versions and increase build success rate?* We integrate the outputs of *BuildCheck* into Spack's concretizer and show an improvement in overall number of packages successfully built. This is additionally demonstrated for a set of packages not included in the training dataset.

2 MOTIVATION

One of the challenges with software components is that each package integrated into an application must be compatible with all other packages. Changes to one package can affect others, and as a result, integration costs grow combinatorially with the package count. Furthermore, the package ecosystem is in a constant state of flux, with packages being added, upgraded, or deprecated. When upgrading a package or its dependency to a newer version, one needs to ensure that the updates are compatible. This typically involves building and installing the package to verify that the changes are valid. However, some updates can trigger a cascade of errors, necessitating an exploration in the package version space to find a compatible configuration.

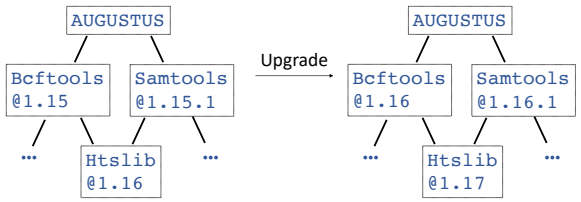

Figure 1: Package upgrade scenario resulting in build errors due to inconsistent versions of packages.

Here is an example showcasing how a change in the version of one package leads to a cascading error. We look at package AUGUSTUS, which is a program that predicts genes in eukaryotic genomic sequences and has 16 dependencies, 3 of which are shown in Figure 1. Suppose the developers of AUGUSTUS want to use the latest version of bcftools, which is 1.16. This requires htslib version 1.17. However, samtools@1.15.1, which is dependent on htslib is not compatible with htslib@1.17. We need to use samtools@1.16.1 to build this successfully. This is known as the *diamond problem* and it is a difficult problem for package managers that manage package versions. To find a compatible version of the software, developers would have to try out different version combinations until they find one that is compatible with all the packages. Then they must add constraints to Spack's package files and submit a pull request back to Spack so that others can benefit from what they learned.

This process becomes impractical when there are hundreds of dependencies requiring considerable time and effort to evaluate compatibility. The build configuration space is combinatorial. For example, the tcl package with just one dependency to zlib contains 98,000 valid configurations when taking into account all possible package versions, compilers, compiler versions, and architectures. We want to identify valid configurations without performing 98,000 builds. For every new architecture and build flag this compatibility search would have to be done manually. In a rapidly changing software ecosystem this is a *constant* effort.

BuildCheck can help with this process. In this example, developers of the AUGUSTUS package can sample the package versions and use *BuildCheck* to predict whether that configuration builds or not. In this way developers can use *BuildCheck* to improve the version constraints of individual packages within their specifications. Furthermore, *BuildCheck* can be integrated into package managers and their version selection mechanisms. This allows the central software provider, the package manager, to reduce the number of failed installs due to version mismatches.

3 BACKGROUND

In this section, we discuss the current state of software build process, present an overview of Spack package management system, and provide sufficient background on Graph Neural Networks.

3.1 Software Build Process

Software packages are highly desirable as they break down complex software into manageable components enabling separation of concern, promoting software reuse, and facilitating installation and updates. Packages are designed by various individuals or communities and their ecosystems enable developers to leverage existing codes to build on top of, thereby saving time and resources. Package management systems are an integral part of any software development process. They provide a convenient way to manage software, enabling users to easily discover, download, and install a wide range of software packages from trusted repositories.

Despite the benefits provided by software components, managing packages can pose several challenges, including handling dependencies and ensuring package compatibility. Moreover, the package ecosystem is in a constant state of flux with packages being added, upgraded, or deprecated. When upgrading a package or its dependency to a newer version, one needs to ensure that the updates are compatible. This typically involves building and installing the package to verify that the changes are valid. However, some updates can cause a cascade of errors necessitating an exploration in the package version space to find a compatible configuration.

Versioning allows developers to declare dependencies on a specific version number or a range of version numbers, which helps package managers determine package compatibility. Using a fixed version can help build a package in a deterministic manner, but it is less flexible and may conflict with other dependent packages making it difficult to use into a large project. Using flexible version ranges allows more customization, but it is not feasible for maintainers to test for all possible combinations. More often than not they assume that the compatibility remains consistent within that range and this can potentially cause a cascade of build errors. While testing can help to a certain extent, it is not possible to evaluate all possible configurations and in the event of a package build failure finding a working configuration may involve significant developer time and system resources.

3.2 Spack

For our work we use the Spack [15] package manager to explore the combinatorial build space of packages and analyze package incompatibilities. Spack is critical for DoE's Exascale Computing Project mission to create robust exascale software ecosystem. It is designed to support building packages from source in a flexible manner to support the various platforms. Spack exposes parameters to users as adjustable knobs and allows a single package to be built in many different ways. This flexibility is critical for HPC ecosystems as HPC developers need scientific software to be built in multiple ways tailored to each platform to obtain optimal performance.

In Spack, package dependencies and constraints are represented as shown in in Figure 2. Package maintainers write constraints like `bzip2@1.0.7:` on line 11 for "bzip version 1.0.7 or higher", or 'zlib@1.2.8:' on line 14 to indicate that the example package requires zlib version 1.2.8 or higher. Constraints can be conditional using a 'when' clause; the `when` clauses here tell Spack that these two constraints only apply for certain versions of the example package. The dependency on `mpi` on line 17 is unbounded; we can attempt to build the example package with any version of any MPI

```
1  # This is the class name for the package `example`
2  class Example(Package):
3      """Example depends on zlib, mpi, and optionally bzip2"""
4
5      version("1.1.0")   # two versions are available
6      version("1.0.0")
7
8      variant("bzip", default=True, description="enable bzip")
9
10     # Depends on bzip2 or later when bzip is enabled
11     depends_on("bzip2@1.0.7:", when="+bzip")
12     depends_on("zlib")  # depends on zlib
13     # Newer versions require newer versions of zlib
14     depends_on("zlib@1.2.8:", when="@1.1.0:")
15
16     # Depends on *some* MPI implementation
17     depends_on("mpi")
18
19     # Known failure when building with intel compilers
20     conflicts("%intel")
21     # Does not support architectures derived from ARM64
22     conflicts("target=aarch64:")
```

Figure 2: Constraints in a Spack `package.py` file, expressed in Spack's embedded DSL.

implementation. The conflicts on lines 20 and 22 indicate that the package does not work on ARM or with the Intel compiler.

The Spack project had over 1,100 contributors at the time of writing, and constraints like those shown in Figure 2 accumulate over time as these contributors add more package information. At the core of Spack is a sophisticated dependency solver called the *concretizer* [14], which finds package configurations that are *valid* in the sense that they satisfy all applicable constraints. However, if the constraints are not fully specified or comprehensive, it may encounter errors when building seemingly valid configurations.

3.3 Graph Neural Networks

Graph neural networks (GNNs) have emerged as a highly-effective technique for analyzing graph-structured data found in various real-world systems, such as social networks, recommender systems, and chemical analysis for drug discoveries. GNNs are able to capture the inherent relationship between nodes in the graph by passing messages along the edges, which are used to generate effective representations. These representations are then used to perform a wide variety of downstream tasks that include node classification, link prediction, and graph classification.

GNNs use both the graph structure and node features to learn a representation vector for each node or for the entire graph. They achieve this by performing neighborhood aggregation, where the node representation is updated iteratively by aggregating its neighbors representations. After a number of iterations, a node's representation captures the structural information within its neighborhood. A popular architecture used in GNNs is the Graph Convolutional Network (GCN) [23]. GCN is a framework of spectral graph convolutions, which is a generalization of convolutions to graph data. It integrates local node features and graph topology in the convolutional layers. Figure 3 shows a GNN for graph classification using Graph Convolutional layers and shows the message passing and state updates.

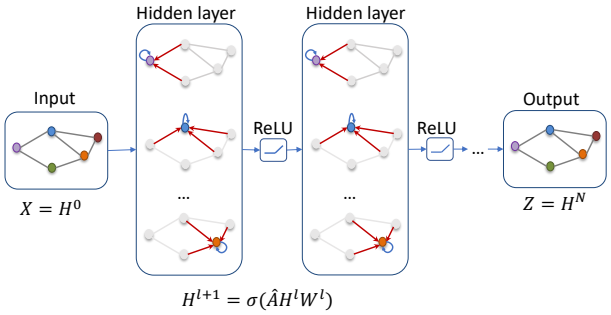

Figure 3: A GNN with Graph Convolutional Network (GCN) Layers. The GNN consists of several layers of graph convolution that operate on the graph data by propagating node features across the graph. The output of each GCN layer is passed through a non-linearity. The output captures the graph structure and is used for the final task.

The update rule for each layer is given by the following transformation.

$$H^{(l+1)} = \sigma(\hat{A}H^lW^l) \tag{1}$$

where H^l is the input matrix to the l-th hidden layer with row H^l_i consisting of a d-dimensional feature vector for node i and W^l is the trainable weights for layer l. $\hat{A}$ is the normalized adjacency matrix. This is passed through a non-linearity function $\sigma(.)$, which in this example is shown as $ReLU(z) = max(0, z)$. The output of the last layer denoted as $Z = H^N$ is then used for the final task. For node classification, softmax(z_n) is used and the output of that is a matrix $\mathbb{R}^{N\times C}$, where N is the number of nodes and C is the number of labels and each row contains a score for each of the labels for that node. In the case of graph classification task, the output of the last layer is aggregated or pooled to summarize the entire graph into a single representative node followed by the application of a softmax given by softmax$(\Sigma_n z_n)$ [11]. The GCN parameters are trained to minimize the cross-entropy error over labeled data.

4 APPROACH

Package management is a crucial aspect of software engineering where compatible versions of packages and their dependencies need to be chosen to ensure that software builds without errors. Current approaches rely on experts with in-depth knowledge of packages and constraints to identify compatible versions. In practice, users often have to explore different choices of package versions to find an appropriate one that builds successfully. Depending on the size of a package dependency graph, this can be time-consuming and error-prone. The problem is exacerbated by the fact that the search space formed by the packages and their versions is large, and exhaustively exploring it is impractical. To address this challenge, we want to use Machine Learning to predict whether a given package dependency graph, with specific versions, can successfully build. Furthermore, we want to automatically come up with version constraints for packages and integrate it with the version selection mechanism. Automating this process can save time and resources for software developers and package maintainers, ultimately resulting in faster and more reliable software development.

4.1 Problem Definition

The package dependency graph is a directed acyclic graph (DAG) that represents the dependencies between different packages. This graph is represented as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes representing the packages and $\mathcal{E}$ is the set of edges capturing the dependencies. Classifying whether a given package dependency graph builds or not is challenging due to the complicated interactions between different packages.

We cast the build success prediction problem as a supervised learning problem. Given a dataset $\mathcal{D} = \{(\mathcal{G}_i, y_i)\}_1^M$ of M graphs $\mathcal{G}_i$ and their corresponding build outcomes $y_i \in \{\text{SUCCESS}, \text{FAILURE}\}$, our goal is to train a prediction model $f(\mathcal{G}; \theta)$ parameterized by the parameters θ. For a given graph $\mathcal{G}$, our prediction model can then be used to predict the build outcome $\hat{y} = f(\mathcal{G}; \theta)$. Supervised learning corresponds to estimating the optimal parameters θ^* using the given dataset $\mathcal{D}$ as following

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{M} L(y_i, f(\mathcal{G}_i; \theta)) \tag{2}$$

Here the loss function $L(y, \hat{y})$ is a measure of the discrepancy between the ground truth label y and the predicted label $\hat{y}$. For discrete prediction tasks, the loss function L is typically chosen to be the cross-entropy loss.

Graph Neural Networks (GNN) are ideally suited for this application and we delve more into the modeling considerations in section 4.2. Further, each sample $(\mathcal{G}_i, y_i)$ in the dataset $\mathcal{D}$ can be quite expensive to obtain as it requires building the entire graph. As a result it is desirable to reduce the data requirements. We consider self-supervised pre-training as a strategy for reducing data requirements in section 4.3. Finally, we show in section 7 how the model can be used to improve version constraints in two software ecosystems and reduce the number of failed builds due to version mismatches.

4.2 GNN model for Package Dependency Graph

Success or failure of a build depends upon the relationships between the packages in the corresponding package dependency graph. To make an accurate prediction for the entire graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, we need to construct a model that can effectively capture and represent the dependency relationships in the graph. In essence, this would require looking at neighboring nodes for each node in the graph and summarizing their interactions. Graph convolutional networks are a promising modeling choice for this task because they iteratively process each node and then aggregate information from neighbors for each node. Note that a graph convolutional layer computes a function of a node and its first order neighbors. To increase the neighborhood, a GCN stacks several graph convolutional layers, where each successive layer increases the order of neighborhoods considered by one.

Our model architecture, shown in figure 4, consists of multiple Graph Convolutional layers. The input to the model is a package specification graph represented by its adjacency matrix A and node features $X \in \mathbb{R}^{N\times d}$ where N is the total number of packages being considered and d is the feature dimensionality. We choose d to be the maximum number of versions for any given package over the set of all packages. Each layer of our model takes a matrix of

features along with the adjacency matrix as input, and outputs another matrix with same number of rows as the input. The output of the GCN layers undergoes normalization through a LayerNormalization layer, followed by the application of the ReLU activation function introduce non-linearity. The output of the final residual block is passed through a linear layer and subsequently pooled globally to obtain a representation of the entire graph, which is used to predict the build outcome. Following is a brief explanation of various layers in our model:

Embedding Layer: This is the first layer of the model and it maps the node indices to a node specific dense representation and combines it with the version features X. Node embeddings represent graph nodes in a continuous vector space. They are a learned dictionary $Z \in \mathbb{R}^{N \times e}$ of e-dimensional embedding per node. The output $\tilde{X} \equiv [Z, X]$ of the embedding layer is a concatenation of the learned node embeddings Z and version features X.

GCN: A graph convolutional layer as detailed in section 3.3.

Residual Blocks: The residual block is a composite building block consisting of several layers and uses an identity shortcut to effectively train very deep neural networks [19]. Our model stacks several residual blocks to construct a deep neural network that can capture higher order neighborhoods.

LayerNorm: A normalization layer that has been found to be effective for stable training of deep models [2].

ReLU: Pointwise non-linearity computed as $\text{ReLU}(x) \equiv \max(x, 0)$.

Pool Layer: This layer computes the average of all node features as a representation for the entire graph.

Linear Layer: This layer computes a linear transform of its inputs and optionally applies the ReLU non-linearity. The last linear layer in the model outputs a two dimensional vector representing the logits for the two possible outcomes {SUCCESS, FAILURE}.

Softmax: Computes the softmax function to yield probability values for the two outcomes.

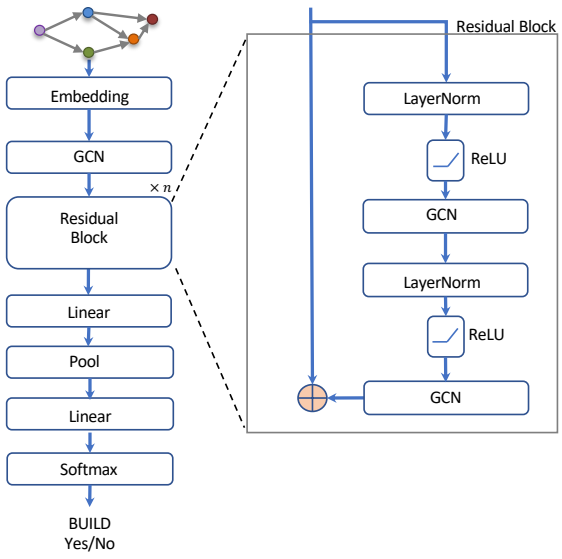

Figure 4: Architecture of our GNN model. Our GNN model is constructed by stacking multiple Graph Convolutional layers in the form of residual blocks.

4.3 Self-supervised Pre-training

The supervised learning of build outcomes requires significant amounts of data that is expensive to collect. Moreover, while it is resource intensive to collect the data, it is also inefficient to train GNNs from scratch for every downstream task. Transfer learning, which separates the training into two distinct steps: 1) pre-training and 2) fine-tuning, has been shown to be very effective [38, 6]. Typically, the pre-training step is performed using any readily available data that may be related to the final task, and does not require labels. Pre-training yields an informative initialization for the model, which can then be fine-tuned in the second step using the labels to yield the final model. More recently, self-supervised pre-training methods have been proposed, where a pretext task is constructed using only the model input data, such that it encourages the model to capture final task relevant characteristics of the data.

The task of predicting whether a package builds or not is based on the compatibility relationship between packages. Therefore, we construct a pretext task using only the input graphs (but not requiring the labels) that encourages the model to capture these relationships. Inspired by the success of masked token modeling in large language models, such as BERT [6], we propose masked build dependency graph modeling as a pretext task. Given a build dependency graph, we "hide" a random subset of packages, and train a model to predict the hidden packages. The intuition behind this choice of pretext task is that the model will be forced to capture the relationships between packages to accurately predict the hidden package and reconstruct the full original dependency graph, given the incomplete version of it. Note that our pretext task does not require build outcomes but only valid dependency graphs, which are much easier to generate.

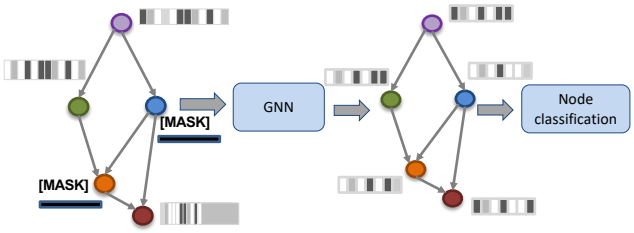

Figure 5: Pretext task training using GNN. For a given build dependency graph, we hide a certain fraction of nodes by replacing their embeddings with a MASK token and train the model to predict the hidden packages.

In practice, given a build dependency graph and a masking fraction, we randomly sample the specific nodes to mask. The nodes are then masked by replacing their node specific embeddings $\tilde{x} \in \tilde{X}$ with a special MASK token embedding, which is also learned. The model then predicts a package for each node. The model is updated based on the loss for masked nodes. Once the model is trained, we apply transfer learning on this pre-trained model to fine-tune for

the downstream task of build outcome prediction. This is done by removing the last linear layer in the pretext task model, adding a pooling layer followed by a linear layer and then fine-tuning the model on the labeled data. Figure 5 shows the overview of our pretext learning task and figure 6 shows a high level overview of transfer learning.

More formally, our pretext task is a classification task. However, instead of predicting the build success, our model now predicts the correct package for each masked node. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where V is the set of packages with $N = |V|$, E is the edges capturing the dependencies between the packages, and $X \in R^{N \times F}$ are the node features, our model computes the embeddings $\tilde{X}$ for all the nodes in the graph. A target fraction of nodes to mask are selected as the set of nodes T, computed as a fraction of total nodes in the graph. Embeddings of the nodes in T are replaced with the embedding for the MASK token. Using these masked embeddings as input, the rest of the model predicts a logit for each of the N possibilities, corresponding to each masked node. The model is trained using the following objective

$$\bar{\theta}^* = \arg\min_{\bar{\theta}} \sum_{i=1}^{K} \sum_{v \in T_i} L(v, \bar{f}_v(\mathcal{G}_i; \bar{\theta})) \tag{3}$$

Here, we use the bar to differentiate between the pretext model and its parameters from the final model and the corresponding parameters. K is the total number of graphs in the dataset and this could potentially be much larger than the number of labeled graphs M. The loss is computed only on the masked set of nodes included in T_i for the graph $\mathcal{G}_i$. Once the pretext model is trained, a subset of the build success prediction model f parameters θ are initialized using the corresponding subset of parameters in $\bar{\theta}$. The reset of the parameters in θ are randomly initialized.

5 EXPERIMENTAL SETUP

In this section, we will provide details of the dataset used in our study and the specifics of the model employed for training on package dependency graphs to predict build outcomes.

5.1 Dataset

We evaluated our model on several packages from the Extreme-scale Scientific Software Stack (E4S). E4S provides open source software packages for developing, deploying and running scientific applications on high-performance computing (HPC) platforms. These software packages are implemented in different programming languages such as C/C++, FORTRAN, Python, Lua, and others. E4S uses Spack for managing software packages. We used the dataset provided in [27] and use it to evaluate our prediction model. These packages are accompanied by their dependency packages, resulting in a total of 367 unique packages in our evaluation dataset. In total we explore 45, 837 unique package builds. Many of these packages have tens and hundreds of dependencies and some of them provide 100 different package versions.

5.2 Implementation

We use PyG (PyTorch Geometric), a library built upon PyTorch, to implement our method. Our model uses Graph Convolutional layers with LayerNorm normalization and RELU non linearity. We train using the AdamW optimizer with a learning rate of $1e-3$ and a weight decay of 0.05. We also use a learning rate schedule with a linear warmup of 50 epochs followed by a cosine decay schedule. Each model is trained for 120 epochs on a single GPU, unless otherwise stated.

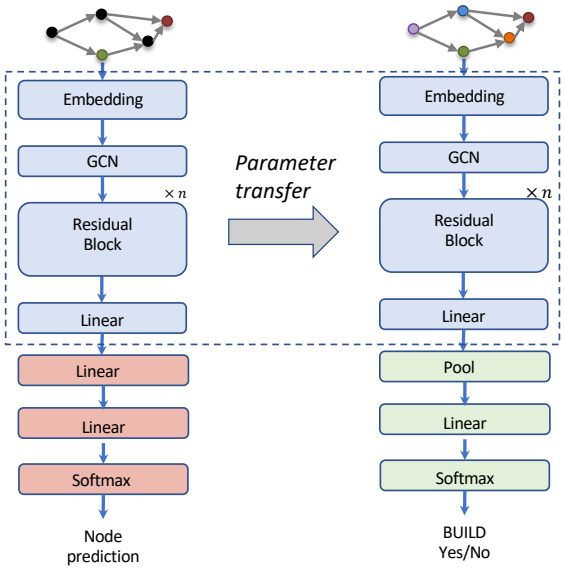

Figure 6: Overview of transfer learning used in our model. knowledge of the pre-trained model is transferred to the final model by initializing it with a subset of parameters from the pre-trained model. This enables us to use the learned representations from the pre-training stage to improve the performance of the final model on the target task.

6 EVALUATION

In this section we present an evaluation of the effectiveness of both the base GNN model as well as the pre-training followed by fine-tuning approach. We begin by evaluating the accuracy for build prediction task and subsequently delve into the effects of different characteristics of the proposed architecture.

6.1 Build Outcome Prediction Evaluation

> **RQ1** *Can* BuildCheck *predict the build outcome of various package configurations with high accuracy?*

6.1.1 Effect of Model Size. We conducted evaluations using a varying number of GCN layers and analyzed their impact on the build prediction accuracy. The results of our analysis for different fractions of training data are shown in Table 1. In general, increasing the number of GCN layers does not improve accuracy significantly and in fact the model performs slightly better when fewer Graph Convolution layers are used. When a GNN model with a single GCN layer is used, the information is aggregated solely over its immediate neighborhood. This means that the model considers only the features of a node and its immediate neighbors to make

	Training Data %					
Layers	0.1%	1%	10%	20%	50%	80%
1	56.54	**74.66**	**84.71**	**87.53**	**90.78**	91.56
3	58.67	74.09	84.10	86.63	90.55	**91.82**
5	**59.49**	74.17	83.82	86.17	90.43	91.41
7	57.69	73.19	83.73	85.89	90.11	91.64
9	58.75	73.52	83.75	86.07	90.53	91.75

Table 1: Test accuracy with respect to different number of GCN layers with a hidden dimension of 256. When only a limited amount of training data is available, for instance at 0.1%, the increase in the number of layers helps the model to perform better. In the absence of sufficient data, the model is able to construct a better representation by relying on information from farther away nodes.

its prediction. As the number of GCN layers in a model increases, the information gets propagated further thereby capturing farther relationships. However, our results suggest that capturing the relationship with farther away nodes does not improve the model's accuracy. Based on this observation, we conclude that the build outcome for the dataset under consideration is primarily determined by pair-wise interaction between a package and its dependencies. When a package and its immediate dependency fails to build, the package fails to build because its requirements cannot be satisfied.

We also evaluated the model's performance using varying amounts of training data ranging from 0.1% to 80% of the entire dataset. We reserved 20% of the entire data for testing. The results of the evaluation are shown in Table 1, which indicate that the accuracy is heavily impacted by the quantity of training data. For instance, when using only 0.1% data, which amounts to 40 examples, the accuracy is notably low at around 59.5%.

Next, we examine the influence of the hidden dimensions on our model. Table 2 shows the effect of the size of the hidden dimensions of a GNN on the graph classification accuracy. If the hidden dimensions are small, then the model may not be able to capture the information and underfit. In contrast, if the hidden dimensions are too large, then the model may overfit to the training data and fail to generalize. Based on our evaluation, we determine that a hidden dimension size of 256 provides an optimal balance between model complexity and generalization performance.

6.1.2 Effect of Residual connections. While deep networks with many GCN layers can capture complex interactions between far-away nodes, it can be hard to train. As the input passes through multiple layers, the gradient can become very small resulting in the vanishing gradient problem. This can cause the model to train poorly. Shortcut connections, also known as skip connections or residual connections, are additional connections that bypass one or more layers of the network allowing information to flow directly from one layer to another without having to pass through intermediate layers. Figure 7 shows the impact of the residual block for different number of layers when trained on 80% of the dataset.

	Training Data %					
#dim	0.1%	1%	10%	20%	50%	80%
32	52.8	71.26	82.85	85.46	88.98	90.99
64	**59.51**	73.07	83.98	86.37	89.97	90.97
128	57.13	**74.21**	83.89	**86.77**	90.42	91.38
256	58.67	74.09	**84.10**	86.63	90.55	**91.82**
512	57.74	74.63	83.45	86.03	**90.95**	91.57

Table 2: Test accuracy with respect to different dimensions for 3 GCN layers architecture. For small hidden dimensions, the model is unable to capture the information resulting in underfitting. For large hidden dimensions, the model is unable to generalize resulting in a decrease in testing accuracy.

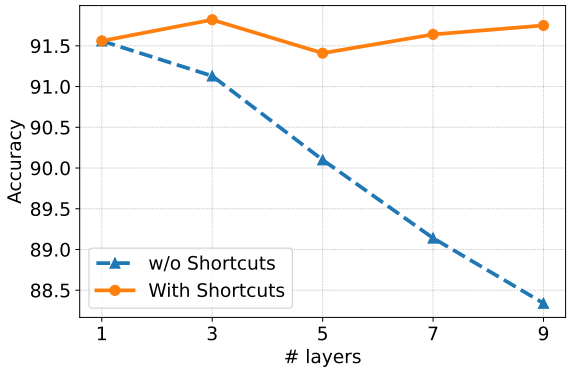

Figure 7: Evaluation of the effect of shortcut or residual connections. Using an architecture with shortcut connections enables the deep models to train better and improve the accuracy by over 3% in the case of a 9 layer model.

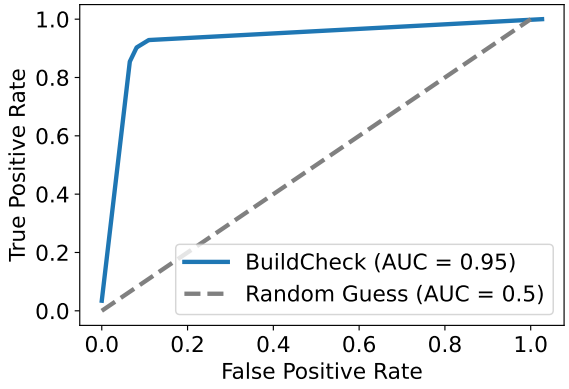

Figure 8: ROC curve showing True Positive and False Positive rate at different classification thresholds. Our model achieves very good AUC of 0.95.

6.1.3 False Positive and False Negative. While false positives result in long, expensive builds that eventually fail, false negatives result in not attempting builds that would succeed. Typically, the set of constraints in a package is incomplete. The constraints are either

too narrow, missing versions that could be built, or too broad such that when the solver comes up with a valid solution it might not build. When the constraints are too narrow, our preference lies in minimizing false negatives and when the constraints are broad our aim is to minimize false positives. In fig. 8, the ROC curve plots the true positive rate (Number of True Positives to Number of Actual Positives) against the false positive rate (Number of False Positives to Number of Actual Negatives) at various classification thresholds between 0 and 1. The model has a high Area Under Curve (AUC) of 0.95 indicating that it is capable of distinguishing between positive and negative outcomes.

6.2 Evaluation of Pretext training and fine-tuning

> **RQ2** *Can self-supervised pre-training be used to reduce the need of expensive build data to train the model?*

We first evaluate the efficacy of the pretraining task and then evaluate the effect of various hyperparameters of the pretext task on the final task after fine tuning.

6.2.1 Effect of Pre-training task. Figure 9 shows the comparison of prediction accuracy between a baseline model and a fine-tuned model using the pre-trained model. The pre-trained model outperforms the baseline model when the training dataset is small. This suggest that using a pre-trained model for fine-tuning on the downstream task using transfer learning can compensate for the limited availability of training data to improve model accuracy. Although the gains become less as the size of the training data increases, it still allows the model to learn faster. Figure 10 shows the training loss of both the models. Since the pre-trained model has already learnt relevant representation, the fine-tuned model starts off with a better initialization, thereby rapidly decreasing the training loss and enabling faster convergence.

6.2.2 Effect of Masking ratios. Figure 11 shows the impact of the masking ratio, which represents the percentage of tokens that are masked during the pre-training of the GNN model. For larger training datasets, a wide-range of masking ratios, from 0.3 – 0.8 work well. For smaller training dataset, the effect of masking ratio is more pronounced. As the masking ratio increases from 0.1 to 0.6, the performance of the model increases, which suggest that masking a large portion of the tokens during the pre-training enables the model to learn a meaningful representation. The model relies on the remaining unmasked token to learn a representation that captures the neighborhood to predict the masked token accurately. However, beyond a 0.6 masking ratio, the model's performance starts to degrade as it lacks sufficient information from unmasked token to learn an effective representation leading to decreased performance.

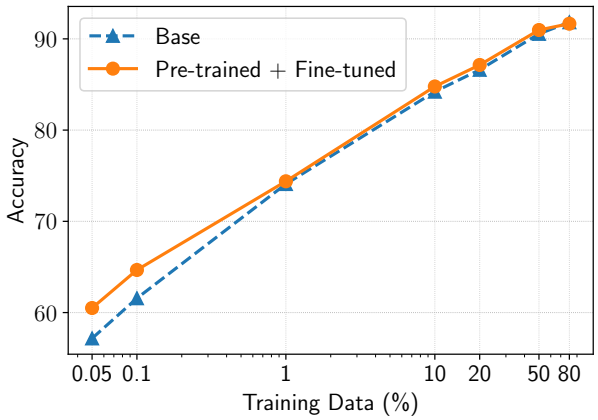

Figure 9: Comparison of the base model with pre-trained + fine-tuned model. The model that has been initialized with the parameters from the pre-trained task performs better than the base model when training dataset is small. For the case of 0.1% and 0.05% of the dataset, which amounts to 45 and 22 examples respectively, the fine-tuned model gives a performance improvement of 3% over the base model.

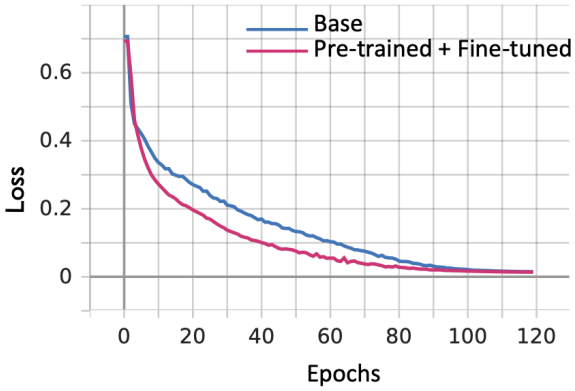

Figure 10: Comparison of loss for the base model and pre-trained + fine-tuned model when trained on 80% of the dataset. Although fine-tuned model does not improve the overall accuracy in the presence of abundant data, it expedites convergence, evidenced by the reduction in training loss in the initial epochs.

7 USING THE GNN OUTPUTS TO SELECT VERSIONS IN SPACK

> **RQ3** *Can the outputs of* BuildCheck *be used to select package versions and increase build success rate?*

In this section we discuss how the outputs of the GNN model can be integrated into the Spack package manager. We further demonstrate how this integration can improve the number of overall successful builds in Spack.

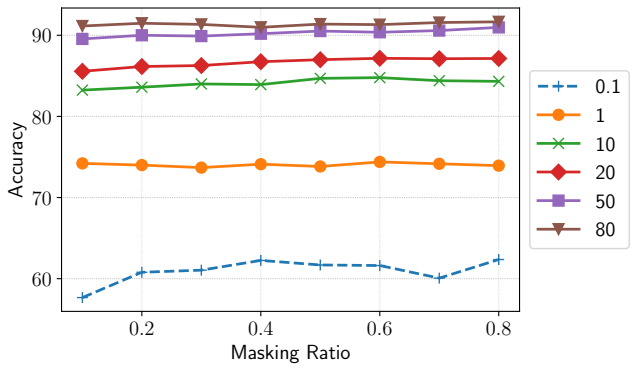

Figure 11: Evaluation of the effect of masking ratio. For larger training dataset, masking ratios don't impact the prediction accuracy. However, when training on small amount of data, masking ratio of 0.40 and above help the model learn meaningful representation that capture the neighborhood to make improved prediction on the final task.

7.1 Integrating Build Probabilities into Spack's Concretizer

Spack's concretizer is responsible for resolving abstract package requirements into concrete dependencies, versions, and build settings. It uses an Answer Set Programming (ASP) solver to find valid configurations given the package and environment requirements. Since there may be a number of valid configurations, Spack also uses several optimizations to find the most optimal configuration. For package versions this involves choosing the package version with the lowest *package version weight*. Packages are weighted 0 for the most recent version, 1 for the second most recent version, etc. Thus, the solver will pick the most recent version of all valid versions for a package. Spack may also select older versions if the user flags the concretizer to reuse existing binaries from the machine or public build cache.

To incorporate the outputs of the GNN model we modify the existing version selection optimization to be a weighted sum of the *package version weight* and *package version pair weight*. The *package version weights* are included in the optimization for packages that are present in valid configuration. Likewise *package version pair weights* are included for parent-child package pairs when they are present in the valid configurations in the solver. Assigning the *package version pair weights* a larger weight in the weighted sum gives the behavior of favoring versions from more likely to build pairs, while falling back to the existing version preferences. For our experiments we found 0.35 and 0.65 to work well for the *package version* and *package version pair weights*, respectively.

The *package version pair weights* are computed as an ordinal encoding of the pairwise build probability of each package pair. Package build probabilities are computed for each parent-child pair for all versions by inputting them as 2 node, 1 edge graphs into the GNN and using the softmax output as its build probability. Then for each pair, all the build probabilities are sorted and assigned an integer encoding based on their index in the sorted list where the pair with the highest build probability is 0. When newer versions of a package exist that are not in the data set we use the probabilities from the nearest pair (based on version distances).

Changing the concretizer to use the weighted sum of version weights is accomplished by a simple change to the minimization criteria in Spack's ASP solver. The *package version pair weights* are incorporated via a separate "facts" file that is included in the solver. This can be generated offline and used across many different installs. It can also be changed between systems to account for potentially different build probabilities on different architectures and OS's.

7.2 Packages for Concretizer Evaluation

We evaluate the new concretizer by building a set of packages from E4S (see Section 5.1). We build each package at all of its available versions in Spack with its default settings. This is done with both Spack's current concretizer and the new one proposed in Section 7.1. Based on these builds we can compare the number of packages successfully built between the two concretizers. Additionally, we can investigate if any packages do not build under the new concretizer that do with Spack's default concretizer.

7.3 Concretizer Evaluation

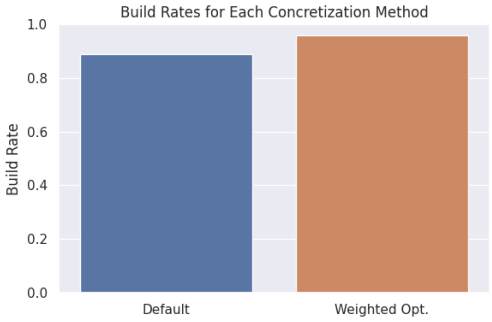

Figure 12: Comparison of build rates between the two concretization methods. The *weighted opt.* concretizer, using the outputs of the GNN, improves the ratio of packages that build from 89% to 96%.

Figure 12 compares the ratio of packages that build successfully for each concretizer. This is the ratio of packages that build successfully to the total number of attempted builds. The new concretizer improves the number of packages that build from 89% to 96% for E4S. The packages that still do not build successfully with the new concretizer are due to compiler vendor/version errors, package definition errors, deprecated dependencies, and unknown linking errors. We removed any results that had errors due to the source repository being taken offline. Additionally, all of the packages that built successfully with the original concretizer built successfully with the new one meaning no new errors were introduced.

8 THREATS TO VALIDITY

Internal Validity: A threat to the internal validity of the study may arise from the GNN model simply correlating entire build graphs or single packages with build success and not understanding how

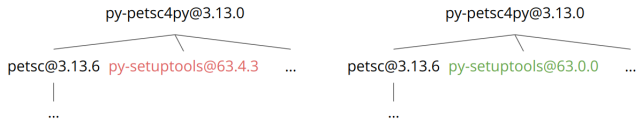

Figure 13: An example improvement using the new concretizer. Changing the setuptools version, which further propagates version changes to other Python packages, leads to a successful build (on the right).

complex packages interplay to create incompatibilities. This threat is first addressed by the training validation tasks. It is also addressed by using independent parent-child pairs inputted into the model to inform actual successful builds in the concretizer.

Dataset quality and specificity may introduce further threats, as build errors may be due to factors beyond version incompatibilities, such as compilers and systems software, and it might not represent broader build errors. To address this and improve model's efficacy for unseen software ecosystems, fine-tuning with a limited number of samples can be performed. Since the model is trained on a substantial dataset of more than 45,000 unique package builds from the E4S software ecosystem, the model is likely to perform well when tested on previously unseen configurations. This is possible because the dependency graph of new recipes will include some packages encountered during the training phase.

External Validity: A threat to the generalizability of *BuildCheck* is that it is only applicable to scenarios where package managers provide support for compiling on a target system. Unlike using a standard Linux package manager, where we are choosing among pre-existing builds that are known to work, in our work, we are choosing configurations to build, some of which may be new untested configurations. Our technique could be applied to maintaining software distributions, such as Debian, RPM, and Redhat.

There is sampling bias present in the selection of packages used for training the GNN. While we believe E4S to contain dependencies representative of all Spack packages, it is possible that the proposed methodologies do not extend to new package sets or package managers. Additionally, the methodology applied to the current data set, which only varies versions, may not produce the same results when different package metadata, such as the compiler, is varied.

9 RELATED WORK

In the 1990s, package management systems emerged as a solution for software installation and management [42], providing mechanisms for downloading, installing, updating, and resolving dependencies. Early package management systems included RPM [12] for Linux, dpkg for Debian-based systems, and apt [17]. Over time, package managers evolved to include sophisticated dependency resolution algorithms. Since the version compatibility problem is NP-complete, the version resolution can be encoded as SAT or Constraint Programming problems [7, 26, 16, 28], an approach that has been adopted by various main-stream package managers [36, 3, 44].

Constraints to the solver are still provided by package maintainers who rely on knowledge regarding version dependencies among packages to detect possible conflicts. Although semantic versioning (*semver*) [35] is commonly used to determine package compatibility based on the version numbers [5, 9], its complex rules are not fully understood [8] resulting in package build breaks [33]. Consequently, developers may switch to using fixed versioning approach, but this can also result in version conflicts and build failtures [13]. Additionally, the study by Kula et al. [24] found that package maintainers are often hesitant to update their dependencies due to the complex inter-dependency relationship, resulting in software systems with known vulnerabilities.

There are several approaches to identify compatible versions of packages. One of the approaches used by maintainers is called the wisdom of the crowd [29], where the most popular or highly used version of a library is chosen. This can prevent build errors to some extent. Some researchers have proposed techniques that automatically change the code to fix the conflicts introduced by their dependencies [45]. Others have proposed tools that identify incompatibilities between different versions of libraries at the binary level [4]. Additionally, there are several works [34, 40, 31, 30, 32, 20] that focus on recommending third-party libraries for use in projects. These amount to adding new dependencies and not necessarily suitable for finding compatible package versions for a given dependency graph. Our goal is to design an entirely data driven method that automatically learns to capture the relevant relationships between the packages that are useful for predicting whether a given package dependency graph builds or not.

A recent work [27] uses Bayesian Optimization to suggest package versions that are likely to build. While that work is useful for picking a configuration that is highly likely to build, it lacks support for reproducible builds and a mechanism to predict success of a specific configuration of interest. In contrast, our approach focuses on predicting whether a given configuration can successfully build, which is a common scenario for package upgrades. Furthermore, the rules derived from *BuildCheck* are incorporated into Spack's concretizer solver, which is used to resolve package contraints, and the solver ensures that it finds an optimal configuration that satisfies minimization criteria, leading to reproducible builds.

GNNs employs message passing across the nodes to learn the structure of the graph dataset. There are various types of GNNs, such as Graph Convolutional Networks [23], Graph Attention Network [43], and GraphSAGE [18], graph autoencoders [39] etc. [22] developed a method for self-supervised pretraining of GNNs. More recently, [37] proposed GCC, a generative pre-training that used contrastive learning and transfers the learnt knowledge to downstream tasks. Masked language modeling, such as BERT [6], has shown to be highly successful for pre-training tasks in Natural Language Processing (NLP) domain. Other self-supervised learning approaches, including GPT [38], have seen a lot of interest with a wide variety of pre-text tasks. However, the use of GNNs and pre-training strategies on Package dependency graph to predict the build outcome has not been done before.

10 CONCLUSION

We have demonstrated how to combine the capabilities of Graph Neural Networks and advanced package management technologies to offer practical solutions for managing package dependencies. Our tool, *BuildCheck*, evaluated on E4S software ecosystem consisting of 45,837 data points can predict build outcomes with 91% accuracy

eliminating very expensive trial-and-error exercise to find working builds. Furthermore, our novel self-supervised pre-training method using masked modeling was shown to improve the prediction accuracy when only a limited amount of data is available. The results of *BuildCheck* make it ready to be used in production for much more reliably successful builds. We use the results from the GNN model with the optimization phase of Spack's concretizer which enables it to steer each solve towards solutions that are more likely to build. We showed that using this for building E4S can help developers and package maintainers to avoid broken configurations and superfluous builds. In conclusion, our results show that using GNNs for predicting build outcomes in conjunction with a package dependency solver can significantly improve software development practices.

11 ACKNOWLEDGEMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-TR-857729). Work at LLNL was funded by the Laboratory Directed Research and Development Program under project tracking code 21-SI-005.

REFERENCES

[1] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques Silva, and Pascal Rapicault. 2010. Solving linux upgradeability problems using boolean optimization. In *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010* (EPTCS). Inês Lynce and Ralf Treinen, (Eds.) Vol. 29, 11–22. DOI: 10.4204/EPTCS.29.2.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

[3] 2014. Cargo: The Rust package manager. Online. https://github.com/rust-lang/cargo. (Mar. 2014).

[4] Bradley E Cossette and Robert J Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 1–11.

[5] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[7] Roberto Di Cosmo. 2005. EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies. Tech. rep. hal-00697463. INRIA, (May 2005).

[8] Jens Dietrich, Kamil Jezek, and Premek Brada. 2016. What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, 21, 3, 1371–1396.

[9] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 349–359.

[10] P. F. Dubois, T. Epperly, and G. Kumfert. 2003. Why johnny can't build [portable scientific software]. *Computing in Science Engineering*, 5, 5, 83–88.

[11] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems*, 28.

[12] Marc Ewing and Erik Troan. 1995. RPM Timeline. Online. https://rpm.org/timeline.html. (1995).

[13] Todd Gamblin. 2021. Software integration challenges. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[14] Todd Gamblin, Massimiliano Culpo, Gregory Becker, and Sergei Shudler. 2022. Using Answer Set Programming for HPC Dependency Solving. In *Supercomputing 2022 (SC'22)*. LLNL-CONF-839332. Dallas, Texas, (Nov. 2022).

[15] Todd Gamblin, Matthew LeGendre, Michael R Collette, Gregory L Lee, Adam Moody, Bronis R De Supinski, and Scott Futral. 2015. The spack package manager: bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12.

[16] Martin Gebser, Roland Kaminski, and Torsten Schaub. 2011. Aspcud: a linux package configuration tool based on answer set programming. *Electronic Proceedings in Theoretical Computer Science*, 65, (Aug. 2011), 12–25.

[17] Jason Gunthorpe. 1998. APT User's Guide. Online. https://www.debian.org/doc/manuals/apt-guide/. (1998).

[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

[20] Qiang He, Bo Li, Feifei Chen, John Grundy, Xin Xia, and Yun Yang. 2020. Diversified third-party library prediction for mobile app development. *IEEE Transactions on Software Engineering*.

[21] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt. 2012. Easybuild: building software with ease. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 572–582. DOI: 10.1109/SC.Companion.2012.81.

[22] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2019. Strategies for pre-training graph neural networks. *arXiv preprint arXiv:1905.12265*.

[23] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*.

[24] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering*, 23, 1, 384–417.

[25] G Kumfert and T Epperly. 2002. Software in the DOE: The Hidden Overhead of "The Build". Tech. rep. UCRL-ID-147343. Lawrence Livermore National Laboratory, (Feb. 2002).

[26] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. 2006. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 199–208.

[27] Harshitha Menon, Konstantinos Parasyris, Tom Scogland, and Todd Gamblin. 2022. Searching for high-fidelity builds using active learning. In *Proceedings of the 19th International Conference on Mining Software Repositories* (MSR '22). Association for Computing Machinery, Pittsburgh, Pennsylvania, 179–190. ISBN: 9781450393034. DOI: 10.1145/3524842.3528464.

[28] Claude Michel and Michel Rueher. 2010. Handling software upgradeability problems with MILP solvers. In *Proceedings First International Workshop on Logics for Component Configuration, LoCoCo 2010, Edinburgh, UK, 10th July 2010* (EPTCS). Inês Lynce and Ralf Treinen, (Eds.) Vol. 29, 1–10. DOI: 10.4204/EPTCS.29.1.

[29] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining trends of library usage. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 57–62.

[30] Phuong T Nguyen, Juri Di Rocco, and Davide Di Ruscio. 2018. Mining software repositories to support oss developers: a recommender systems approach. In *IIR*.

[31] Phuong T Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. Crossrec: supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 161, 110460.

[32] Phuong T Nguyen, Juri Di Rocco, Riccardo Rubei, Claudio Di Sipio, and Davide Di Ruscio. 2021. Recommending third-party library updates with lstm neural networks.

[33] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2021. Breaking bad? semantic versioning and impact of breaking changes in maven central. *arXiv preprint arXiv:2110.07889*.

[34] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M German, and Katsuro Inoue. 2017. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83, 55–75.

[35] Tom Preston-Werner. 2013. Semantic versioning 2.0. 0. (2013).

[36] Python Software Foundation. 2020. New pip resolver to roll out this year. Online. https://pyfound.blogspot.com/2020/03/new-pip-resolver-to-roll-out-this-year.html. (Mar. 2020).

[37] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. Gcc: graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 1150–1160.

[38] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1, 8, 9.

[39] Arindam Sarkar, Nikhil Mehta, and Piyush Rai. 2020. Graph representation learning via ladder gamma variational autoencoders. In *Proceedings of the AAAI Conference on Artificial Intelligence* number 04. Vol. 34, 5604–5611.

[40] Zhensu Sun, Yan Liu, Ziming Cheng, Chen Yang, and Pengyu Che. 2020. Req2lib: a semantic neural model for software library recommendation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 542–546.

[41] Chris Tucker, David Shuffleton, Ranjit Jhala, and Sorin Lerner. 2007. OPIUM: optimal package install/uninstall manager. In *International Conference on Software Engineering (ICSE)*.

[42] Andre Van Der Hoek, Richard S Hall, Dennis Heimbigner, and Alexander L Wolf. 1997. Software release management. *ACM SIGSOFT Software Engineering Notes*, 22, 6, 159–175.

[43] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat*, 1050, 20, 10–48550.

[44] Natalie Weizenbaum. 2018. PubGrub: Next-Generation Version Solving. https://medium.com/@nex3/pubgrub-2fb6470504f. (Apr. 2018).

[45] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: inference and application of api migration edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 335–346.