Regulating Traffic in a Crowded Cache: Overcoming the Container Explosion Problem

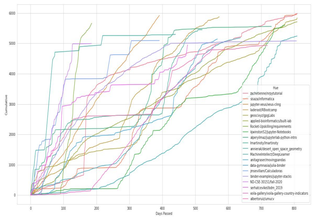
Kevin Gao, Tim Shaffer (Advisor), Kyle Chard (Advisor)
University of California, Berkeley, University of Notre Dame, University of Chicago

Abstract

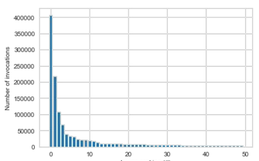
- **Containers are used by multi-user online services to deploy customized computing environments for users**
- **There is often significant overlap between users' environments and thus sharing and caching containers can improve response time and performance**
- **We explore caching approaches leveraging various metrics (e.g., package size, installation time, popularity) for services that create Python environments.**
- **We design a simulator that evaluates custom heuristics and compares their performance to an LRU Cache implementation**
- **Using production workloads from Binder, we show that our methods can reduce total storage consumption by *1–3%* and container creation time by *6–11%* when compared with a least recently used strategy.**

Binder

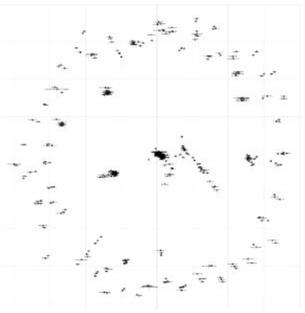
- Binder is an online service that allows users to execute Git repositories and run interactive notebooks
- Binder dynamically creates a container for each launch based on the dependencies specified in the repository



An initial high riser: launch invocations over time for 20 repositories over a 600 day period. Most repositories tend to show a large spike of usage at the beginning while a few have a balanced distribution of invocations throughout their lifetime.



Frequency of launches for each unique container image. We see a long tail distribution where few containers are launched many times, while most containers are launched infrequently.



Clustering of Python packages based on cosine similarity. We see visible "islands" of commonly related packages.

Our Dataset

We filtered the Binder launch dataset to only Python repositories

- *14 million launch records*
- *4575 unique Python libraries*
- *33,987 unique repositories*
- *17,019 unique images*

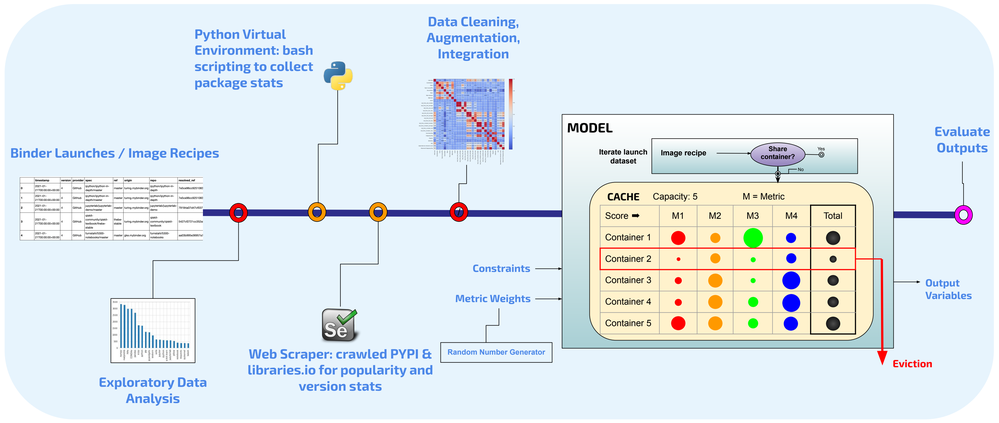
For each repository we obtained the Python dependencies and computed

- *Install time and size*
- *GitHub statistics (stars, forks)*
- *Version lifetime (how long between versions)*

Methodology

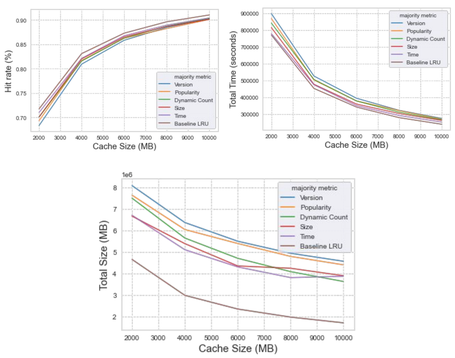
Simulator

- Loops through each binder launch temporally
- Considers:
  - Version mismatch
  - Constraints
- Assigns a score to each container based on combination of metric values
- Implements container sharing and a lowest score caching policy



Heuristic Evaluation

- **Version**: *average(version count) / min(current date - date of first version)*. How often a package version changes.
- **Popularity**: *average(stars / median(stars) + forks / median(forks))*. A combination of package popularity (GitHub) statistics.
- **Size**: *sum(size)*. Total package size on disk.
- **Time**: *sum(time)*. Total package install time.
- **Dynamic Count**: *average(dynamic freq) / (# of launches)*. An online approach that counts the number of package invocations within a sliding window.



A comparison across different metrics and how they perform against **LRUCache (None)** individually. Without a combination of multiple heuristic types, any single heuristic performs poorly on average against the baseline LRU model. This difference is reflected most significantly in the total container storage usage, as the baseline beats any metric by roughly 150-250% depending on cache size.

Weighted Combination

A "holistic" score assigned to a container:

Let M = iteration over each metric
m_coeff = weighting for metric, which is randomly generated from a Gaussian distribution

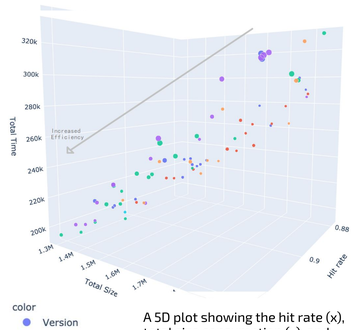
1. Naïve: *sum(m_coeff * M / median(M))*
2. CacheRank (CR): *sum(m_coeff * Rank(M))*
3. CR w/ standardization: *sum(m_coeff * (Rank(M) – mean(Rank(M)) / std(Rank(M))*

Prioritizing the removal of containers from the unprotected part of the cache:

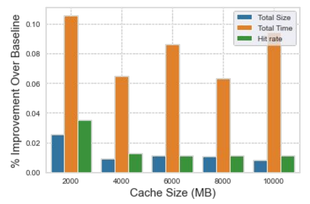
1. Score: *min("holistic" score)*
2. Size: Minimum Knapsack algorithm to greedily maintain the largest cache size possible
3. Score + Size: *min("holistic" score / container size)*.

Extremely small containers will have a large score and be hard to remove.

The average total size consumption of each simulation at an intersection between score and removal priority methods. The naive holistic score is expected to be weakest, but achieves admirable efficiency when paired with the combination method in removal priority.



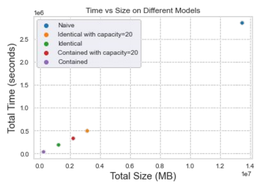
A 5D plot showing the hit rate (x), total size consumption (y), and total installation time (z). The cache size is represented by the size of a circle and the most dominant metric type (largest absolute value) is the color of the plotted point.



Percentage improvements for total size, time, and hit rate shown between the most optimal set of parameters and base LRU model. The greatest improvement is shown in the total installation time, with a >6% improvement at every tested level of the cache size constraint.

Container Sharing

1. Baseline: No containers are shared. Requests must use a unique container for each repository.
2. Identical: Repository with the same packages may share containers.
3. Contained: Repositories with a subset of packages from another repository may share that container.

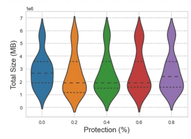


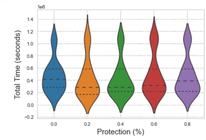
Version Compatibility

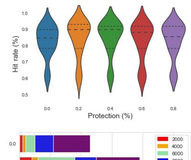
For each container there is a possibility that it is incompatible when package versions are conflict. While this reduces opportunities for sharing, it is more realistic when considering reproducibility.

MRU Cache Protection

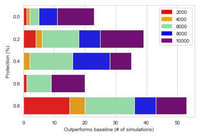
- First *x* containers in LRU are "protected" from eviction
- Mixed strategy to prioritize most recently used items at the front of the cache and apply metrics to sort the back of the cache







We used every 20% as the percentage to protect in the cache. We see that this strategy provides useful insight as the strongest average results tend to come from protection values in the middle, [0.4, 0.6, 0.8].



Number of containers that outperform the base LRU model at every protection level, where color represents the cache size constraint. While the highest protection value is strongest at low cache size limits, the middling values get stronger as the cache limit increases.

References

[1] Binder, https://mybinder.org, 2021
[2] T. Shaffer, K. Chard, D. Thain, "An Empirical Study of Package Dependencies and Lifetimes in Binder Python Containers". eScience, 2021.
[3] T. Shaffer et al., "Solving the Container Explosion Problem for Distributed High Throughput Computing," IPDPS, 2020

Acknowledgements

This work was supported in part by the National Science Foundation grant NSF- 1461260 (BigDataX REU) and NSF-2004894 (funcX)