# Forklift: Fitting Zygote Trees for Faster Package Initialization

Yuanzhuo Yang
University of Wisconsin-Madison
Madison, WI, USA

Kwangjong Choi
University of Wisconsin-Madison
Madison, WI, USA

Keting Chen
University of Wisconsin-Madison
Madison, WI, USA

Tyler Caraza-Harter
University of Wisconsin-Madison
Madison, WI, USA

## Abstract

Fast cold start is critical for many serverless applications. For Python, startup is typically dominated by module initialization. In this work, we analyze the dependencies of 9,678 Python applications and create ReqBench, a new benchmark for stressing module initialization. Informed by our findings, we design Forklift, a new algorithm for training zygote trees based on invocation history. Each zygote pre-imports some modules and can be forked to create other zygotes or function instances. We integrate Forklift with OpenLambda, improving median invocation latency by about 5× while using only a modest memory footprint (<6 GB).

## CCS Concepts

• **Computer systems organization → Cloud computing**.

## Keywords

function-as-a-service, dependency support, zygote initialization

**ACM Reference Format:**
Yuanzhuo Yang, Kwangjong Choi, Keting Chen, and Tyler Caraza-Harter. 2024. Forklift: Fitting Zygote Trees for Faster Package Initialization. In *10th International Workshop on Serverless Computing (WoSC '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3702634.3702952

## 1 Introduction

As serverless platforms [1, 2, 4, 5] become increasingly popular, fast startup becomes ever more important. Recently, there has been significant progress towards reducing the startup latency of containers [13], virtual machines [6, 8, 16, 19], and unikernels [10, 18, 23]. Lightweight sandboxing, however, is only half of the solution; the other half is initializing processes inside these sandboxes. Popular languages such as Python and JavaScript have corresponding package repositories. Tools such as pip and npm make it easy for developers to construct applications that build upon libraries in these repositories; the libraries, in turn, frequently depend on other libraries. Unfortunately, importing these resources introduces significant startup latency [20]. When many applications have the same dependencies, these startup costs are paid repeatedly.


*WoSC '24, December 2–6, 2024, Hong Kong, Hong Kong*

ACM ISBN 979-8-40007-1336-1/24/12
https://doi.org/10.1145/3702634.3702952

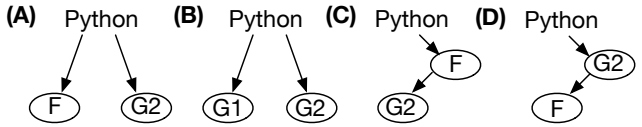


**Figure 1: Example Zygote Trees. Nodes correspond to processes, labels indicate imported packages, and arrows point from parents to forked children.**

One popular solution is zygote initialization [3, 7, 9, 11, 17, 20]. With this approach, a zygote process pre-imports frequently-used modules, but does not run any specific application. Applications needing those modules provision the processes by creating copy-on-write clones of the zygote. This approach is *fast* (child processes already have resources imported), *efficient* (physical memory containing code is shared across different processes), and *isolated* (processes attempting to modify shared pages trigger copy on write).

In this work, we focus specifically on hierarchical zygotes [20]. When zygotes are hierarchical, parent zygotes importing an initial set of packages are used to provision child zygotes, each of which import additional packages and may have their own children. The resulting structure is a zygote tree. The central question of this paper is: *how should a large tree of zygotes be organized*?

Figure 1 shows that there are numerous possibilities, even for a trivial scenario involving two packages, F and G. Is it better to allocate memory for one version of each package (tree A), or only support two releases of a popular package G (tree B)? The answer clearly depends on workload patterns. Zygote tree structure must also reflect dependencies between packages. For example, if package F has a dependency on G2, then tree D is reasonable, but C is not.

Assuming no dependency relationship between packages F and G2, we must consider security when selecting between trees C and D. We assume public repositories are not carefully vetted for malicious packages or security bugs [22]. Developers are responsible for selecting safe dependencies for their functions, and serverless platforms should not initialize function processes from zygotes that import dependencies beyond those selected. Thus, a function requiring only F can safely use tree C, but the zygote for F in tree D exposes the function to an unwanted package, G2.

Our first contribution in this work is a detailed study of 9,678 Python requirements.txt files from GitHub (§2). This study reveals dependency patterns with numerous implications for the construction of zygote trees. For example, the top 15 packages appear in over 50% of the files, so relatively few zygotes could provide substantial benefit. The average application only has 7 direct package dependencies but 24 total package dependencies; the indirect nature of dependency layers is a natural fit for hierarchical zygotes. Moreover,

about half of package dependencies are flexible regarding what version of a package to use, creating opportunities to select versions based on available zygotes. We also contribute a new benchmark, ReqBench (§3), based on our dataset of dependencies.

Based on our package study, we design a new algorithm, Forklift (§4), for generating efficient zygote trees based on historical invocation data. The Forklift algorithm is inspired by classic algorithms for iteratively constructing decision trees, one node at a time. We adapt the OpenLambda serverless platform [20] to support Forklift trees. This requires building new support into OpenLambda for multiple package versions. We also introduce a lazy-loading optimization so workers can start quickly, even with large trees. We use ReqBench to carefully evaluate Forklift tree structure and performance benefits (§5). We find that trees importing multiple packages in a single node outperform trees with single-package nodes. The best trees improve invocation latency by 5× while consuming <6 GB of RAM. We finally discuss related work (§6) and conclude (§7).

## 2 GitHub PyPI Dependency Study

Python package installation typically involves using *pip* to install packages from the PyPI repository. For instance, *pip install p==1.2.3* installs a specific version, while *pip install -r requirements.txt* installs multiple packages listed in the given file. A requirements.txt file may only list direct dependencies, with *pip* identifying indirect dependencies recursively. In this section, we analyze requirements.txt files to discern Python package usage patterns. Understanding these patterns is crucial for constructing efficient zygote trees.

Our analysis utilizes the *GitHub Activity Data* from the BigQuery public dataset [12]. This dataset offers a snapshot of over 2.8 million open-source GitHub repositories as of November 26, 2022, encompassing commits, file paths, and contents. Specifically, we extracted 9,678 unique requirements.txt files from repositories updated post-April 21, 2022 for our study.

We expect most of these applications are not meant to be deployed as serverless functions. However, understanding and benchmarking a broader class of applications is useful; doing so may enable us to build the next generation of serverless platforms capable of running applications that are a poor fit for current platforms.

We explore several questions in this section. *How precisely do programmers specify package versions* (§2.1)? *How many direct and indirect packages do applications typically require* (§2.2)? And *what is the popularity distribution across packages* (§2.3)?

### 2.1 Version Specification Patterns

A requirements.txt file may or may not specify version requirements for a package; when it does, the requirement may be exact (*i.e.* ==) or less precise (*e.g.* !=, =, <, <=, >, and >=). We categorize version specification patterns as follows: *Implicit*: there is no version requirement given, so `pip` can select a version; *Explicit (latest)*: an exact version was specified, and that version was the most recent, as of the dataset's last update; *Explicit (others)*: a non-latest version was specified; and *Range*: other version requirements involving comparison operators.

Figure 2 shows how versions are specified for the 15 most popular packages, as well as an average over all 11,842 packages in our dataset. For the average package, the most popular specifications are *Explicit (latest)* and *Implicit*, at 40.3% and 35.6%, respectively. The average package uses *Explicit (others)* only 9.8% of the time,

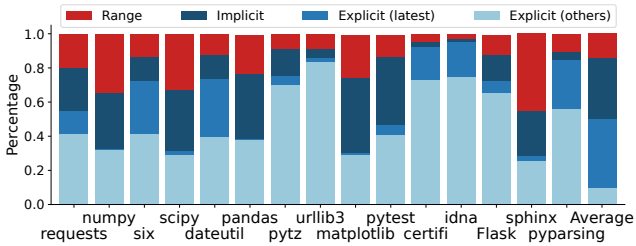


**Figure 2: The version specification breakdown for the 15 most popular packages. The comparative "Average" bar represents the percentages for the entire dataset.**

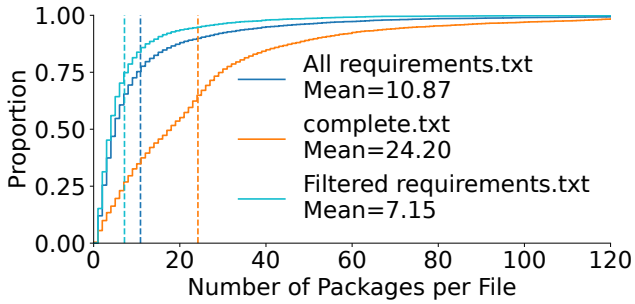


**Figure 3: CDF plot showing the number of packages per file type.**

but explicit versions are used more often with popular packages, perhaps because established packages have a longer history of releases to choose from.

**Implications**: For the average package, Python applications are flexible about what version is used (*Implicit* and *Range* options represent the majority of cases). This creates an opportunity for execution platforms to choose versions based on what packages are already pre-installed (or otherwise warm) on a server.

### 2.2 Requirement Counts

We now explore how many package dependencies, direct or otherwise, a typical Python application has. Indirect dependencies are packages that are not directly required by the project but by the project's dependencies. In order to infer (a) precise package versions and (b) indirect dependencies, we attempt to *pip-compile* [21] each requirements.txt file in our dataset to produce an output requirements file (complete.txt) with precise versions of all packages (direct or otherwise). The output of a *pip-compile* depends on the environment and the latest package versions at the time of execution. We generated the complete.txt files with Python 3.10 on September 21, 2023. Among our 9,678 requirements.txt files, 71% (6,870) were compiled successfully. Failures occurred due to inconsistent file formats, version updates, *etc.*

Figure 3 shows a CDF of dependency counts for raw requirements.txt files, complete.txt files, and the subset of the requirements.txt files on which pip-compile ran successfully. An average requirements.txt file only has 11 requirements, but the compilable ones were somewhat shorter (about 7). For a typical application, over 70% of dependencies are indirect (the complete.txt files are, on average, 3.4× longer than their corresponding inputs). 10.63% of the complete.txt files have >50 packages.

**Implications:** Most package requirements are indirect, so package initialization may be costlier than one might expect; fortunately, zygotes can help alleviate these costs. The multi-layered nature of package dependencies is a natural fit for hierarchical zygote trees.

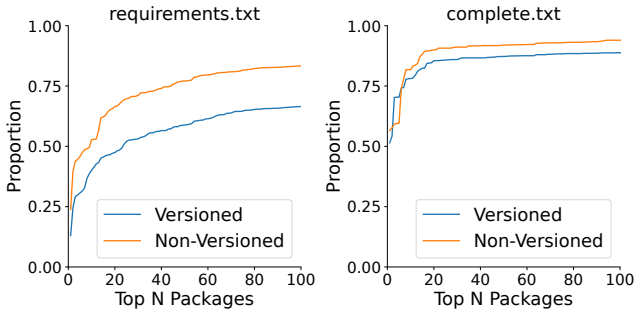


**Figure 4: The cumulative coverage plot depicting the frequency with which the top X packages appear in the requirements files relative to the total data set.**

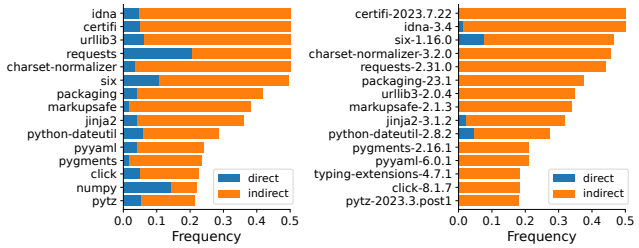


**Figure 5: The stacked bar plot delineating the proportion of direct versus indirect package usages in complete.txt files.**

## 2.3 Popularity Distribution

Some packages are more "popular" than others (*i.e.*, they are dependencies for more applications). We now explore the popularity skew of the packages in our dataset.

Figure 4 shows how many requirements.txt files specify at least one of a set of popular packages. The x-axis indicates the number of top packages considered, and the y-axis displays the percentage of all requirements.txt files where these top packages appear. Remarkably, the top 15 packages alone account for more than 50% of the files for both requirements.txt and complete.txt.

Figure 5 shows how often the top 15 packages appear in complete.txt files, both when grouping together all versions of the same packages (left) or treating each package version as distinct (right). We observe that the 6 most popular packages are used as dependencies in about half of all applications. We also observe that the most popular packages are usually indirect dependencies.

**Implications:** The significant skew in package popularity indicates that relatively few zygotes could provide substantial benefit.

# 3 ReqBench

We introduce ReqBench, a new serverless benchmark designed to measure the efficiency of module import time and memory consumption. We construct a suite of ReqBench functions from our dataset of complete.txt files (§2) as follows: identify the $p$ most popular packages, select complete.txt files that use only those packages, and generate a function corresponding to each complete.txt file. Each function is a no-op, importing only the top-level modules of direct dependencies (85% have a single top-level module).

The ReqBench executor invokes functions in parallel and is extensible for different FaaS platforms. Note that we cannot vet the security of all the packages of ReqBench (malicious packages have been found in the PyPI repository [22]), so any execution or installation of ReqBench functions should be sandboxed.

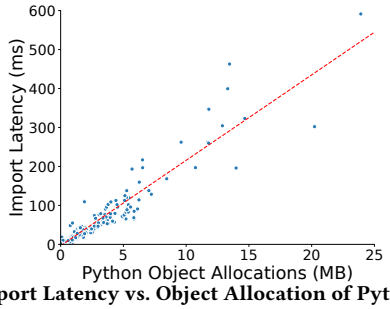


**Figure 6: Import Latency vs. Object Allocation of Python Modules, with linear regression fit.**

ReqBench exposes $p$ and $n$ parameters. *Package Count (p)*: select the $p$ most popular packages during construction of the function suite. The default setting ($p = 500$) yields 1793 functions. *Concurrency (n)*: ReqBench tasks issue requests as fast as necessary to keep $n$ requests outstanding at all times (default: $n = 5$).

**Workload Properties:** We download the 500 packages for ReqBench's default mode; prior to installation, the cumulative compressed size of these packages is 492.9 MB. Installing all the predownloaded packages sequentially takes 4.9 minutes. The installed packages have an on-disk footprint of 1.8 GB. We sequentially import each package's top-level modules (excluding 22 packages for which imports fail due to faulty dependency metadata); the imports take 12.7 seconds. We measure the time and memory cost (measured as wall-clock time and the cumulative size of Python objects, respectively) of importing the top-level modules for each package. Figure 6 shows a scatter of the 500 packages used. We observe a strong correlation between memory allocation and time ($R^2 = 0.91$). We also observe that 43% of packages take over 9 ms to import. This is a substantial cost, given that a typical application has 7 direct (and 24 indirect) package dependencies (§2.2).

# 4 Forklift Zygote Trees

Our analysis shows that a typical Python application requires ~24 packages when counting indirect dependencies (§2.2) and importing modules for these packages is costly (§3). High package popularity skew (§2.3) suggests relatively few zygotes could alleviate initialization work for many functions. Furthermore, popular packages frequently serve as dependencies for many different "higher-level" packages; hierarchical zygote implementations better reflect these relationships than flat zygote systems. In this section, we first introduce Forklift, an algorithm for training zygote trees on historical invocation data (§4.1). Second, we describe how we integrate Forklift trees with the OpenLambda [14] serverless platform (§4.2).

## 4.1 Tree Construction

We propose a new approach for constructing zygote trees from historical function-invocation data. We first describe the basic algorithm that treats all packages equally and constructs trees with one package per node, where the zygote process corresponding to each node pre-loads the top-level modules of that package. We then optimize the algorithm to (a) prioritize "heavy" packages and (b) allow nodes to load multiple packages.

Algorithm 1 describes the basic version of the Forklift algorithm. The algorithm trains the tree on historical invocation data, represented as a binary *calls matrix*. The basic version supports different

**Algorithm 1** Forklift Tree Construction: Basic Version

```
1:  candidateQ ← PriorityQueue() // highest utility first
2:  function BUILD_TREE(calls, desired_nodes)
3:      root ← Node(calls, no packages)
4:      ENQUEUE_TOP_CHILD_CANDIDATE(root)
5:      while desired_nodes > 0 and len(candidateQ) > 0 do
6:          best_candidate ← candidateQ.pop()
7:          ADD_CHILD_NODE(best_candidate)
8:          desired_nodes − −
9:      return root
10: function ENQUEUE_TOP_CHILD_CANDIDATE(parent)
11:     candidates ← empty list
12:     for child_pkgV in parent.calls.column_names do
13:         if child_pkgV is valid then
14:             utility ← sum(child_pkgV column of parent.calls)
15:             add (parent, child_pkgV, utility) to candidates
16:     candidateQ.push(highest-utility candidates entry, if any)
17: function ADD_CHILD_NODE(candidate)
18:     parent ← candidate.parent
19:     child_pkgV ← candidate.child_pkgV
20:     child_calls = rows in parent.calls that import child_pkgV
21:     child ← Node(child_calls, child_pkgV)
22:     parent.add_child(child)
23:     update parent.calls to remove child_calls rows
24:     ENQUEUE_TOP_CHILD_CANDIDATE(parent)
25:     ENQUEUE_TOP_CHILD_CANDIDATE(child)
```

versions of the same package, and each package/version combination is represented as a column in the matrix. Each function invocation is represented as a row, and 1's indicate the packages a function being invoked needs (including indirect dependencies).

The *BUILD_TREE* function starts with a single-node tree, then repeatedly adds nodes to the tree until the tree is a desired size. Each node (except the root) indicates what package the zygote should pre-load. A process started from a zygote will have all the packages pre-loaded along the path from the root zygote to the zygote that is forked. Thus, a package may often be assigned to multiple nodes in the tree to provide fast initialization for different package combinations. Each node has its own calls matrix, indicating which function invocations would be routed to that zygote for initialization. Whenever a child is added, a subset of the parent's rows are taken away and used to create the child's call matrix.

At any given time, we can add many child nodes to the existing tree nodes. A function, *ENQUEUE_TOP_CHILD_CANDIDATE*, identifies the best potential child for a given node, according to a utility measure, and places that candidate in a global priority queue, *candidateQ*; the main loop of the algorithm pulls from this queue. The utility of a candidate is computed as the sum over the column corresponding to the package/version that the candidate's zygote would pre-load; in other words, utility (for now) is simply a measure of usage frequency. The *ENQUEUE_TOP_CHILD_CANDIDATE* function is called on any new node added to the tree, such that each node in the tree always has an associated candidate, which is the best potential child that could be added to to that node.

The *ENQUEUE_TOP_CHILD_CANDIDATE* function is also responsible for suppressing some invalid candidates. For example, a node N should not be responsible for a package P unless the ancestor nodes of N are responsible for the dependencies of P. Also, a package that has already been imported by an ancestor should not be considered again as a candidate. Finally, conflicting versions of the same package are not allowed.

The *ADD_CHILD_NODE* is responsible for actually turning a candidate into a new child node in the tree and moving some of the parent's rows to the child. *ENQUEUE_TOP_CHILD_CANDIDATE* is then called to find the best candidate for the new node. There is no limit on the number of children a node may have, so a new candidate is also found for the parent of the new node (the new node was previously a candidate, so after it is added, the parent needs a new best candidate).

**Optimizations**: The final Forklift algorithm has two improvements beyond the basic approach. First, we profile packages and give more weight to those with slow module imports. One could alternatively weight based on memory consumption, though these two measures are highly correlated (§3). We implement priority by replacing the 1's in the binary calls matrix with the weight values (recall that utility is a sum over a column in the matrix).

For our second optimization, we remove the requirement that a node can only have one corresponding package/version. We do so by lifting the requirement that a node N can only be responsible for P if N's ancestors are responsible for P's dependencies. When we assign P to N, we simply assign any of P's unsatisfied dependencies to N as well. The utility is then calculated by summing the weights of all dependencies and multiplying this total by the number of calls to import these packages.

## 4.2 Tree Deployment in OpenLambda

We adapt the OpenLambda serverless platform [14] to use Forklift trees to initialize containers for functions. OpenLambda is based on SOCK [20], a lightweight container implementation that supports a sandbox-level fork. Although this mechanism supports hierarchical zygotes, the SOCK approach has two major limitations: (1) there is no support for different versions of the same package and (2) zygotes are created greedily to serve the needs of the current invocation, without consideration of historical workload patterns.

We modify the OpenLambda worker to support multiple package versions. In our version, functions must provide complete requirements.txt files listing exact versions of every package (users may generate these using *pip-compile* on a regular requirements.txt file). All package versions installed by workers are visible via package mounting inside every sandbox; automatically configured path variables ensure functions import the desired version. Unless function code tampers with the path, functions are not exposed to potentially malicious unrequired packages.

We also modify OpenLambda to take a tree specification upon startup. In a typical deployment, we imagine Forklift running in the background (*e.g.*, hourly), generating trees based on recent invocations. OpenLambda workers could restart when there is a new tree. To speed up restart, zygotes are created lazily upon first use. Zygotes may be evicted under memory pressure.

Upon an invocation, OpenLambda traverses the zygote tree, starting from the top and choosing the deepest zygote that only

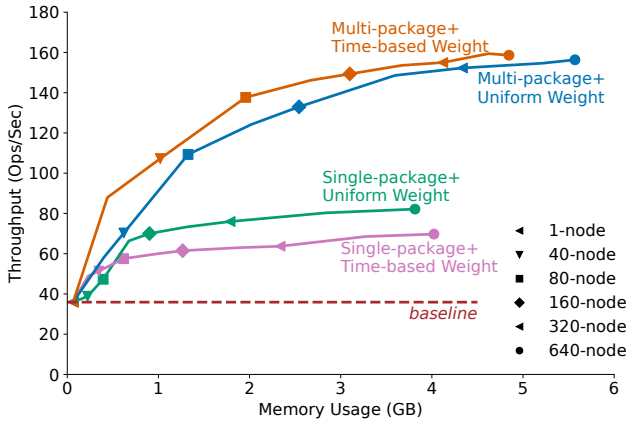


**Figure 7: Memory usage and throughput for different Forklift trees.**

includes requested packages (if a zygote Z provides a package a function F does not need, it would be insecure to initialize F from Z, as packages are neither vetted nor trusted). If multiple zygotes match the needs of the function, the left-most child is selected.

## 5 Evaluation

We use ReqBench to evaluate Forklift trees on an OpenLambda worker deployed on an Azure VM (8 cores, 32 GB RAM, Ubuntu 22.04). To avoid overfitting, we split the default ReqBench call trace, which consists of 1793 functions (each with a single invocation), into equal sized train and test traces. Forklift constructs trees from the train trace, and we execute the test trace to measure performance.

Kernel bottlenecks from cgroup locking and namespace unsharing hindered deploying large trees under heavy load. Therefore, we disabled unsharing and configured OpenLambda to reuse cgroups in our experiments. This setup reduces isolation between functions, but the performance results better represent the benefits of zygote trees if future kernel work (orthogonal to our work) removes these concurrency bottlenecks.

For the first experiment, we construct trees of different sizes, using four variants of the Forklift algorithm. We measure workload throughput and zygote consumption for each tree at steady state (achieved by playing the trace twice and measuring performance during the second run). Throughput matters directly to cloud providers (high-throughput workers save money) and often indirectly to users (providers may pass some savings along to customers). Figure 7 shows the results, averaged over 10 runs. The best tree improves throughput by 4.3× relative to the baseline (a single zygote consisting of an initialized Python interpreter and no additional modules). All the trees have a modest memory footprint (<6 GB of RAM). We observe that assigning multiple packages to a single zygote is a critical optimization; the trees that do so double throughput relative to their single-package equivalents. Finally, weighting packages by import time is helpful for smaller trees, but the benefit becomes less significant for the largest trees.

Figure 8 show a CDF of invocation latency for a single run of the workload. We see that both small and large trees provide large speedups relative to the baseline (a single zygote). The baseline's median latency is 76.5 ms; with small and large trees, median requests are 3.2× and 4.8× faster, respectively. At the 95th percentile, speedups are 2.7× and 5.3×, respectively.

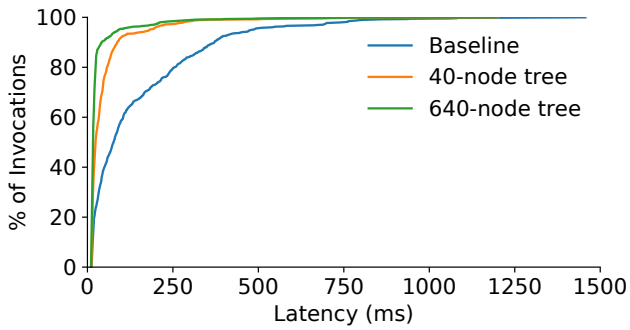


**Figure 8: Latency CDF For multi-package time-based Forklift tree.**

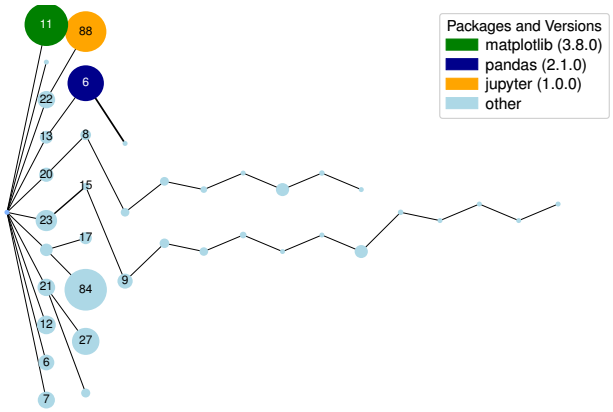


**Figure 9: Multi-package, Uniform Weight 40-node tree.**

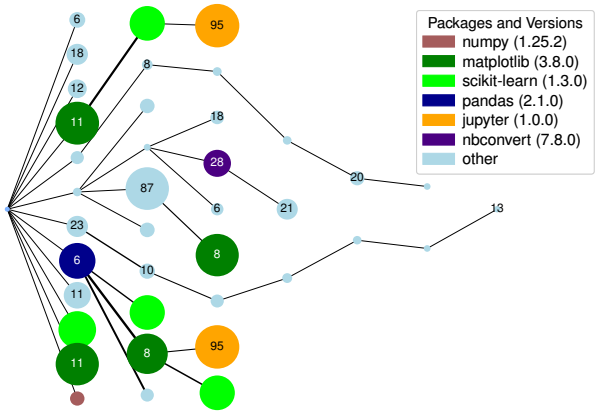


**Figure 10: Multi-package, Time-based Weight 40-node tree.**

Figures 9 and 10 show small (*i.e.*, 40-node) multi-package trees, without and with time weighting, respectively. We observe that for small trees such as these, the uniform-weight version (which only considers frequency) misses many of the heavy zygotes that the time-weighted variant constructs. Larger trees will naturally serve more package combinations, so for that case it is less critical to add nodes for heavy packages first. The nodes in the figures are annotated to indicate how many packages are pre-loaded. For example, the Jupyter nodes usually require about 90 packages. Clearly, initializing a 90-node zygote chain (from root to leaf) will be costly; anecdotally, this explains why the single-package zygote trees perform poorly relative to the multi-package trees.

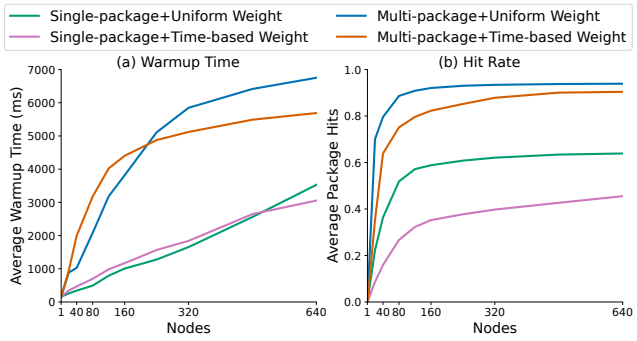


**Figure 11: The average warmup time and average packages hit rate.**

Although we initialize zygote processes lazily upon first use, some providers may wish to eagerly initialize zygotes to make the performance more predictable. Figure 11a shows that all zygotes can be created in less than 7 seconds, even for large trees (we concurrently create the zygote processes with six threads).

Figure 11b shows hit rates for trees of different sizes. When a function is initialized from a zygote, we count every package the zygote provides to the function as a hit; packages not provided by zygotes are misses. We see that the multi-package, uniform-weighted tree has the best hit rates (over 90%); the fact that the time-weighted tree is the fastest indicates that not all misses are equal (some package imports are slower than others).

## 6 Related Work

Many systems use zygotes for fast initialization. Java processes on Android [9] are an early case (and the first usage of the term "zygote" of which we are aware). More recently, Meta open-sourced Cinder [3], a modified Python interpreter that optimizes garbage collection for copy-on-write memory and is used for zygote initialization by Instagram. Fast startup is especially important in serverless settings; OpenLambda [20], SAND [7], Catalyzer [11], and Pagurus [17] have all previously used zygote-based techniques to initialize serverless function instances. Prior work focuses on zygote mechanisms, whereas Forklift orthogonally suggests a new way to fit zygote trees to historical invocation data.

Pre-initialized instances are an alternative to zygotes [19]. While pools can improve latency, zygotes are more efficient: zygotes can support high throughput whereas pools may become exhausted, and zygotes naturally support memory sharing between instances. Sharing does entail risk: address space randomization is difficult to implement in combination with zygotes [15], but not with pools.

Dependencies are not the only cause for slow startup. Firecracker [8] and Kata [6] are recent lightweight virtual machine implementations designed for fast startup. RunD [16] (based on Kata) is designed to overcome some of the challenges we encountered with cgroup bottlenecks.

## 7 Conclusion

We have analyzed the package requirements for over 9,678 Python projects on GitHub and found that modern applications rely heavily on third party packages, which in turn rely on other packages. We introduce Forklift, a new algorithm for constructing trees of zygotes to help initialize these dependencies more quickly for new sandboxes. We have integrated Forklift trees with OpenLambda, making invocation latency about 5× faster for the ReqBench workload.

## 8 Acknowledgements

We thank Remzi H. Arpaci-Dusseau and the anonymous WoSC reviewers for their valuable feedback.

## References

[1] 2024. AWS Lambda. https://aws.amazon.com/lambda/.
[2] 2024. Azure Functions. https://azure.microsoft.com/en-us/services/functions/.
[3] 2024. Cinder. https://github.com/facebookincubator/cinder.
[4] 2024. Google Cloud Functions. https://cloud.google.com/functions/docs/.
[5] 2024. IBM OpenWhisk. https://developer.ibm.com/openwhisk/.
[6] 2024. The speed of containers, the security of VMs. https://katacontainers.io/.
[7] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*. 923–935.
[8] Jeff Barr. 2018. Firecracker – Lightweight Virtualization for Serverless Computing. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.
[9] Dan Bornstein. 2008. Dalvik VM Internals. https://www.kandroid.org/board/data/board/AndroidBeginner/file_in_body/1/2008-05-29-Presentation-Of-Dalvik-VM-Internals.pdf.
[10] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15.
[11] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.
[12] GitHub. 2022. *GitHub Repos Public Dataset.* https://console.cloud.google.com/marketplace/product/github/github-repos
[13] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 181–195. https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter
[14] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.
[15] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. 2014. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 424–439.
[16] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 53–68.
[17] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 69–84.
[18] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 559–573.
[19] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233.
[20] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX annual technical conference (USENIX ATC 18)*. 57–70.
[21] The pip-tools Contributors. 2023. pip-tools documentation v7.3.0. https://pip-tools.readthedocs.io/en/stable/
[22] Nikolai Philipp Tschacher. 2016. *Typosquatting in Programming Language Package Managers.* Ph. D. Dissertation. Universität Hamburg, Fachbereich Informatik.
[23] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 173–186.