Pipsqueak: Lean Lambdas with Large Libraries

Edward Oakes
University of Wisconsin-Madison
oakes@cs.wisc.edu

Leon Yang
University of Wisconsin-Madison
guohong@cs.wisc.edu

Kevin Houck
University of Wisconsin-Madison
houck@cs.wisc.edu

Tyler Harter
Microsoft Gray Systems Lab
tyharter@microsoft.com

Andrea C. Arpaci-Dusseau
University of Wisconsin-Madison
dusseau@cs.wisc.edu

Remzi H. Arpaci-Dusseau
University of Wisconsin-Madison
remzi@cs.wisc.edu

***Abstract*—Microservices are usually fast to deploy because each microservice is small, and thus each can be installed and started quickly. Unfortunately, lean microservices that depend on large libraries will start slowly and harm elasticity. In this paper, we explore the challenges of lean microservices that rely on large libraries in the context of Python packages and the OpenLambda serverless computing platform. We analyze the package types and compressibility of libraries distributed via the Python Package Index and propose PipBench, a new tool for evaluating package support. We also propose Pipsqueak, a package-aware compute platform based on OpenLambda.**

1. Introduction

Cloud computing has democratized scalability: individual developers can now create applications that leverage thousands of machines to serve millions of users. Beyond simple scalability, though, many modern web applications also require elasticity, the ability to scale quickly. Highly elastic applications can rapidly respond to load changes, both saving money under light traffic and taking advantage of flash crowds and other opportunities [6, 11].

Elasticity depends on fast deployment. An application will not be able to gracefully serve a sharp load burst if it must first provision new virtual machines and perform lengthy software installations.

Deployment stresses every type of resource: code packages are copied over the network, packages are decompressed by the CPU, the decompressed files are written to disk, and application code and state must be loaded into cold memory. All of these costs directly correlate with the size of the deployment bundle; a smaller bundle will require less network and disk I/O, will be faster to decompress, and will require less space when loaded into memory, resulting in a more elastic application.

Developers are partly responsible for creating small deployment bundles, but there is much that cloud providers can do to facilitate the development of lean applications. In particular, platforms can encourage sharing and the decomposition of applications into smaller components. For example, Figure 1 contrasts a traditional cloud platform based on virtual machines with a serverless platform, such as OpenLambda [10]. Deployment of new instances of the application on virtual machines will be slow because the application logic is tied to the runtime and operating system. Starting a new instance will require copying the virtual-machine image to a new physical machine, booting the operating system, and loading the execution runtime (*e.g.*, a language virtual machine, such as the JVM) into memory.

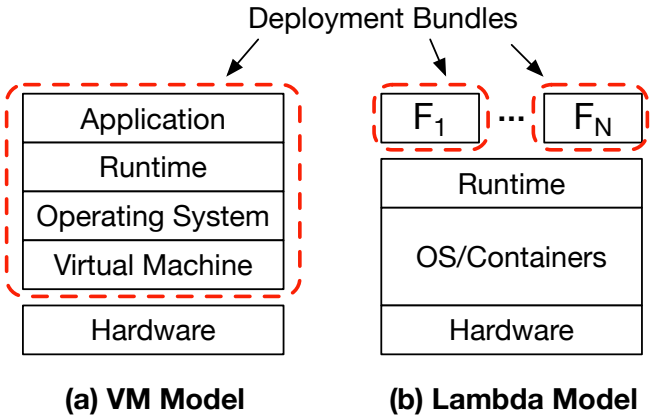

Figure 1. **Compute Models.** *A cloud service based on virtual machines (left) is compared to a serverless platform based on Lambdas (right).*

In contrast, Figure 1b shows how a Lambda-based platform encourages developers to shrink their deployment bundles along two dimensions. First, developers are encouraged to reduce the vertical size of their application by building on top of standard shared components (*e.g.*, specific Linux kernels and runtime environments) which can then be pre-initialized, ready to be used by any application. Second, the Lambda model forces developers to write their application as a set of handlers that run in response to events. Thus, even though the whole application may be large, individual handler bundles will be smaller and faster to deploy.

Unfortunately, Figure 1b shows a simplistic use case

where the Lambda handlers do not depend on user-space packages. In practice, developers rely on an assortment of libraries to avoid implementing everything from scratch. If a handler is distributed along with its dependencies, a conceptually lean function will have to be deployed in a large bundle. Executing the function on a new machine may require copying said bundle, decompressing it, writing the contents to local disk, and loading it into memory. It is easy to see how such costs could dominate the latency of a Lambda invocation. Of course, asking developers to eschew the use of popular packages is not acceptable. Such requirements will surely deter adoption of serverless computing.

In response to the problem that large libraries pose for serverless microservices, we propose Pipsqueak, a package-aware serverless platform based on OpenLambda [10]. Pipsqueak will cache packages from the Python Package Index (PyPI), the primary Python repository, by maintaining a pool of Python interpreter processes as cache entries, each of which has a set of packages installed and already interpreted. Each process will act as a template from which to clone new, pre-initialized interpreters to serve requests. We explore several new policy factors that must be considered in this new type of cache. For example, potentially unsafe packages impose new constraints on cache-entry selection, and the use of copy-on-write memory between processes means that an evictor will need to consider state shared between processes.

The rest of the paper is organized as follows. First, we further motivate the need for a package-aware Lambda platform (§2). Next, we describe the PyPI repository as well as its associated package management tool, pip. (§3). We then propose Pipsqueak, a package-aware platform based on OpenLambda (§4). Finally, we suggest PipBench, a new tool for evaluating performance of handlers with external dependencies (§5) and conclude (§6).

2. Motivation: The Library Challenge

Decoupling an application from its operating system in order to increase sharing can reduce deployment sizes by an order of magnitude [14]. Unfortunately, modern applications rely heavily on large libraries [8] and other userspace dependencies [9]. Bundling these dependencies with each Lambda handler leads to bloated deployment packages and very slow response times [1].

Figure 2a illustrates the problem. Even though the developer split the application into small functions (F_1 to F_N), dependencies on large third-party libraries such as `numpy` and `scipy` dominate the handler sizes. During a load burst, these packages will need to be copied and installed to many worker machines.

One solution would be to rewrite old packages, splitting the functionality of these large dependencies into many smaller bundles. For example, perhaps the numerical `numpy` package could be refactored as a set of Lambda handlers (*i.e.*, one for fast Fourier transform, one for matrices, *etc.*) and used via inter-Lambda REST calls. Mass deployment of these monolithic libraries could be replaced with on-demand deployment of only the features that are actually used.

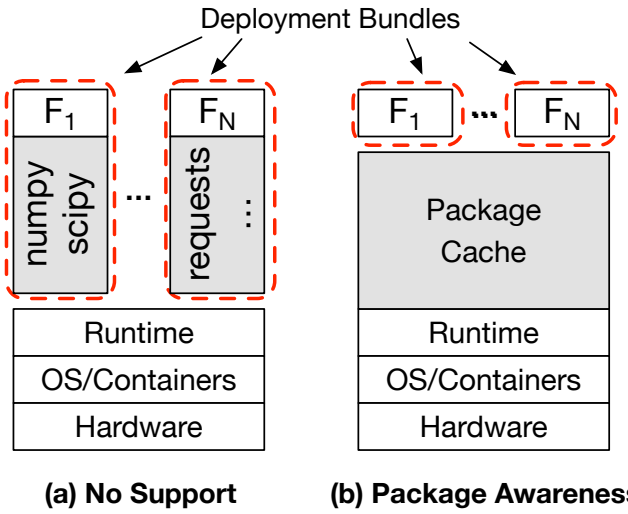

Figure 2. **Library Support.** *Without special library support, packages will need to be installed in each Lambda handler (left), creating large deployment bundles. With package support (right), a common repository of packages can be shared between different handlers. The shaded boxes represent packages.*

Unfortunately, refactoring the packages of all the popular language repositories would be no small task, and relying on Lambda-specific implementations limits the flexibility afforded to developers.

Instead, we propose building package support as part of the serverless platform, as shown in Figure 2b. In this design, the platform would support a set of language-specific libraries (*e.g.*, the PyPI repository) and track which are required by each handler. This type of dependency awareness would allow the serverless platform to share packages between handlers belonging to different customers. It would also allow a subset of requests to be handled by workers with required packages in a hot state (*i.e.*, already installed, and possibly in memory) for increased performance. In a later section, we describe our plans to implement support for the PyPI repository in OpenLambda (§4).

3. A Study of Python Packages

Most modern scripting languages now have large repositories of popular packages and tools for easily installing packages and their dependencies [17]. Ruby has RubyGems, NodeJS has npm, and Python has PyPI. In order to understand common package characteristics and usage patterns, we plan to analyze the Python packages distributed via the PyPI repository. Toward this end, we have set up a PyPI mirror, downloading a copy of the entire repository.

Figure 3 shows the total size of the PyPI packages (as of March 19, 2017); simply downloading the packages requires 466 GB. However, most packages are compressed as *.tar.gz* files or with a zip based format (*.whl*, *.egg*, or *.zip*). In uncompressed form, the cumulative size of the packages is 1.3 TB. We observe that the simple *.tar.gz* packages are more popular than the Python-specific *.egg* and *.whl* files. Across the bars, the number of compressed subfiles is about 100×

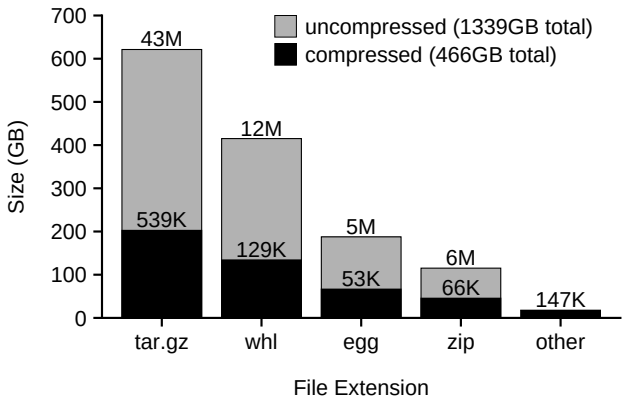

Figure 3. **PyPI Mirror.** *The logical size of a PyPI mirror (excluding directory entries) is shown, compressed and uncompressed. The sizes are broken down by file type. The number of files in a category are shown at the bar ends. Data was collected on March 19, 2017.*

greater than the number of files, indicating that a typical packages will contain about 100 files.

Implications: The PyPI repository is too large to cache in memory on every worker machine in a serverless cluster. Uncompressed, the packages are over 1 TB, which is also too large to store on SSDs on every worker at low cost.

4. Pipsqueak: Package-Aware Lambdas

In this section, we propose Pipsqueak: a new package caching mechanism in OpenLambda [10]. We first describe the different levels of caching that are possible (§4.1) and discuss security requirements (§4.2). We then describe the basic mechanism for caching (§4.3) as well as new policy decisions to be made at the local and cluster levels (§4.4).

4.1. Startup Costs

If a package is being used by a handler for the first time (*i.e.*, there is no cached state to rely on), the following steps will be necessary:

Download: Fetching the compressed packages from a repository mirror in the cluster may be necessary. This will consume both network bandwidth and SSD resources on the worker. It is conceivable that all the packages could be stored locally on every worker compressed (466 GB), but using that amount of SSD capacity would be costly.

Install: Packages are normally compressed, so they will need to be decompressed and written to disk before they can be used. Furthermore, some packages (*e.g.*, numpy) have extensions written in C, so installation may require a compile phase. Installation is guided by a `setup.py` script that the developer provides, and this script may execute arbitrary code, so there may be other steps we do not describe here that are specific to individual packages.

Import: Importing a module within a package involves executing the `__init__.py` script in the root of the directory, often involving defining functions and importing other modules. Thus, there will be a CPU cost to generating Python bytecode. Furthermore, `__init__.py` may execute arbitrary code, including calls into other languages such as C.

In order to be able to quickly execute a new handler, we would like to have its necessary packages already downloaded, installed, and imported. Our measurements (§3) suggest it is not practical to have every package initialized in this way on every worker machine, so a caching policy will dictate which packages are pre-initialized.

4.2. Security Assumptions

We assume that handler code may be malicious. We further assume that PyPI packages may be malicious. In practice, Tschacher [17] showed that it is easy to upload malicious packages to the most popular repositories for Python (PyPI), NodeJS (npm), and Ruby (RubyGems). While one could imagine vetting packages that are included in such a repository, doing so on such a large and rapidly growing body of code would be nontrivial.

With respect to the steps described earlier (§4.1), we assume downloading packages is safe since it only involves copying files. We assume installation and importing, however, may be malicious, as these steps may involve executing arbitrary code submitted by a malicious user. These assumptions lead to three design decisions:

1) Package installation and importation must always be performed in a sandboxed (*e.g.*, containerized) environment in order to protect the host worker.
2) In order to protect users from malicious packages, a handler H must never be allowed to run in an environment where package P has been imported or installed, unless handler H depends on P.
3) We provide no protection guarantees to a handler that chooses to import a malicious package. For example, it is acceptable for information to leak between the handlers belonging to different customers if the handlers import the same malicious package. Note that this problem is not unique to a serverless computing environment [17].

4.3. Cache Mechanism: Interpreter Forking

Our goal is to able to quickly provision a new Python interpreter for a handler, pre-initialized with the packages the handler needs downloaded, installed, and imported into memory. More generally, we need to acquire a mostly initialized process without paying the cost of starting a new interpreter and loading a variety of dependencies. Thus, we plan to build our interpreter cache as a collection of paused Python processes, each with a different set of packages already imported. Using a cache entry will simply involve calling fork from a cache entry to allocate a new, pre-initialized interpreter process to run the handler code.

This basic design is complicated by security concerns. First, user code may be malicious, so when the engine forks a new process from a cache entry, that new process will have to join a new container. Second, packages are also assumed to be malicious, so cache entries will also need to be isolated

in containers. Thus, a newly forked interpreter will need to be horizontally relocated from one container to another. The steps to provision a pre-initialized interpreter and use it to run handler code in service of an event E will be as follows:

1) Select a cache entry; let P_{parent} and C_{cache} be process and container corresponding to the entry
2) Wake up the P_{parent} process and signal it to fork a new child interpreter process, P_{child}
3) Allocate a container $C_{handler}$
4) Map the handler-specific code into $C_{handler}$
5) Relocate P_{child} from C_{cache} to $C_{handler}$
6) Forward the event E to process P_{child}

We plan to implement the above logic using containers in a Linux environment. In Linux, a container consists of a set resource limits (applied to sets of processes called *cgroups*) and namespaces. The APIs for moving a process to a namespace or cgroup in Linux resemble the following:

```
// set up namespaces:
setns(namespace_fd, ...);
// set up resource limits:
write(cgroup_fd, <PID>, ...);
```

In the above code, both the `namespace_fd` and `cgroup_fd` descriptors are obtainable with `open()` calls to various paths in a pseudo file system.

One advantage of the cgroup interface is that it can be applied to an arbitrary process. Thus, the OpenLambda engine running on the host can move any process to any cgroup, without cooperation from the process being moved.

In contrast, the `setns` (set namespace) call moves the *calling process* to the specified namespace, so the engine cannot call `setns` on behalf of another process. Thus, the newly forked process will be in a container, so it will not be able to obtain descriptors to another container without assistance. We plan to work around this by implementing a capability-passing mechanism that allows the OpenLambda engine to provide a newly forked process in a caching container with a descriptor, which it can then use to move itself to a handler container.

Figure 4 illustrates how our design works as it provisions a handler H that relies on packages A and B. The engine first provisions a container for the new handler, and obtains file descriptor references to the namespaces and cgroups for that container. It then selects a cache entry that already has packages A and B imported. It sends a "move" signal to the cache-entry process, which will fork a child, and use the file descriptors from the engine to move the child to the newly provisioned container. Observe that some cache entries (*e.g.*, the "A,B" entry) may have been provisioned from parent cache entries that had a subset of packages already imported (*e.g.*, the "A" entry).

Security: We assume packages may be malicious. The cache-entry interpreter may have already imported one or more packages, and those packages may have modified our in-container wrappers to retain references to the cache-entry container. Such behavior would undermine data isolation between different customers initialized from the same cache

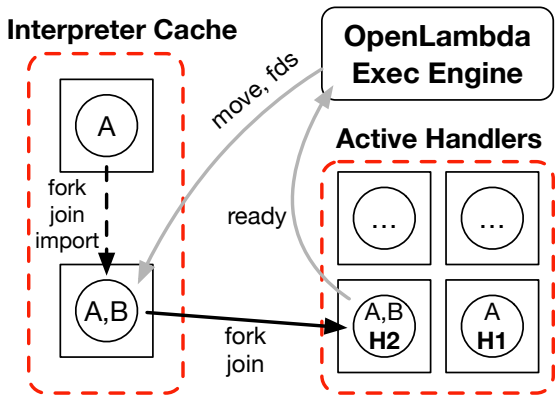

Figure 4. **Interpreter Caching Mechanism.** *Circles represent processes, solid rectangles represent containers, and dashed rectangles represent collections of containers. A "join" operation consists of entering the namespaces and cgroups of the destination container, and it is assumed that the "A,B" entry has already been created via the dashed arrow operations.*

entry. This outcome is acceptable, though, as we make no guarantees in the case that a customer chooses to import a malicious package. This means that it is only safe to use a cache entry if the packages imported by the entry are known to be safe or the handler chooses to use them.

4.4. Cache Policy: Tree Management

In the previous section, we described a mechanism for caching initialized modules in sleeping Python interpreters. In this section, we discuss the broad issues that must be considered when implementing a corresponding policy.

Tree Cache: The engine will fork a cached process in order to obtain an initialized Python interpreter with certain packages already imported. Sometimes a cache entry may be forked from another cache entry; in this case, the child is able to import further packages in addition to those imported by its parent. The use of fork creates parent/child relationships between cache entries. Within the kernel, this means that copy-on-write will enable sharing of physical pages between processes, reducing memory costs [4, Ch 23]. The use of processes as cache entries in this way means that the cache will resemble a tree, mirroring the process tree, as shown in Figure 5. This structure has implications for candidate selection and eviction.

Candidate Selection: Suppose (in the context of Figure 5) that a handler is invoked that requires packages A and B. It is tempting to select Entry 4 to use as the template for our new interpreter; this would be fastest because all the requisite packages are already imported. However, what happens if package C is malicious? In this case, we expose the handler to code that it did not voluntarily import. Unless packages are vetted for safety, we must select a cache entry that has imported a subset of the packages needed by a handler. Note that partial performance benefits could be achieved by partial vetting: if a cloud provider vetted some packages (in this case C) but not others (*e.g.*, X), it would be possible to use Entry 4 in our example.

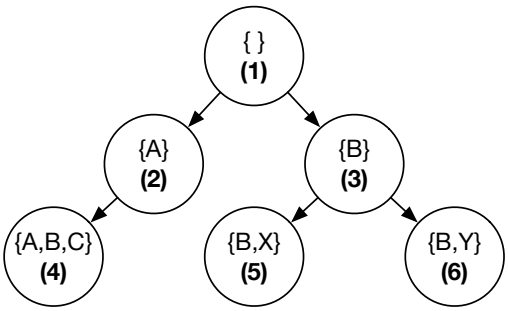

Figure 5. **Tree Cache.** *Circles represent cache entries, which are sleeping processes. Sets of letters indicate the packages imported by a process; parenthetical numbers are used to identify an entry. Arrows represent the parent-child relationships between processes that result from our use of forking.*

If we suppose that packages are not vetted, cache Entries 2 and 3 are still reasonable candidates. Choosing between multiple viable options could be guided by performance metadata that describes typical install and import times for each package. This would allow the engine to select the interpreter template that has already completed the most initialization work.

Eviction: Eviction policies typically select entries to evict as needed to reclaim resources on an individual basis. Unfortunately, such local decision making will often be insufficient in the context of a tree cache. Suppose (again in the context of Figure 5) that the engine is considering entries 3, 5, and 6 for eviction. Further suppose that Package B is very large, but Packages X and Y are both small. The memory consumed by B is shared between the three entries, so evicting only one or two of the entries will not completely reclaim that space. This leads to a scenario where the amount of space that would be reclaimed by evicting all three entries together will be greater than the sum of the reclaimed space from evicting each entry individually. We expect similar situations to arise frequently in practice, rendering greedy eviction algorithms ineffective.

Global Scheduling: Our discussion so far has focused on the cache tree of a single worker machine. Of course, in a real cluster deployment, there will be many worker machines, each of which will have its own tree cache. A global load balancer will need to decide which workers should handle which requests. To do so effectively, an effective balancer must consider the local cache state of worker machines in addition to monitoring load distribution [13].

Maintaining such knowledge at the balancer level is non-trivial. In addition to knowing which packages have recently been required by a given worker, a balancer would benefit from awareness of local caching decisions. For example, if a worker has recently used package X, did the cached interpreter import other packages at the same time? If so, the subset-only security rule means that certain handlers using X will not be able to benefit from the cache entry. Deciding what cache information should be shared between workers and balancers will be a key research challenge for package-aware platforms.

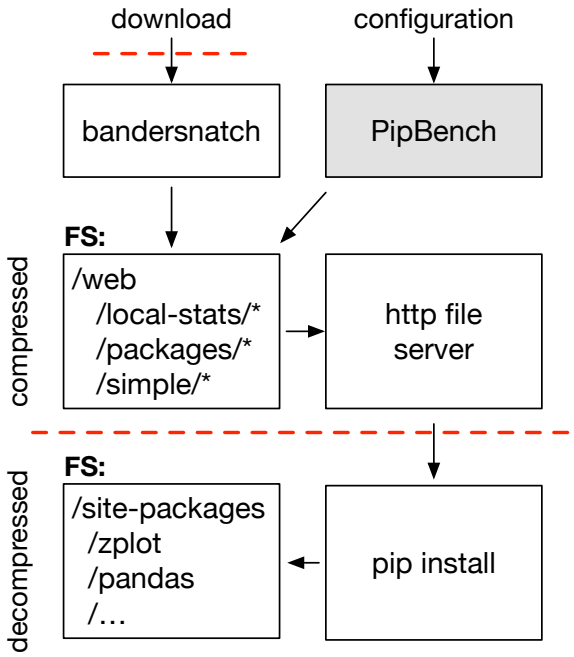

Figure 6. **Pip Workflows.** *The flow of packages from the web, to a local mirror, and finally to a client is shown. Dashed lines represent network boundaries. The shaded box represents our planned contribution.*

5. Evaluation: PipBench

In order to evaluate library support in Pipsqueak and other package-aware microservice platforms, new benchmarking tools are required. Towards that end, we propose PipBench, a new tool for generating artificial packages and workloads that utilize said packages. PipBench will consist of two pieces: a server and a client. The server will act as a PyPI package mirror [2] containing artificial packages, and the client will generate requests to Lambda handlers which import the artificial packages in the mirror.

Figure 6 illustrates a typical flow of packages from the web to a client that is performing an install. First, a pip mirroring utility called *bandersnatch* executes regularly, fetching packages from the web, and storing them in compressed form to a directory in the local file system. The directory tree populated by bandersnatch may then be hosted by a generic HTTP server. Pip clients pull packages from the server, decompress them, and write them to the local file system of the client.

One option in designing the server piece of PipBench would be to take a snapshot of an actual PyPI mirror. Unfortunately, such snapshots are unwieldy (hundreds of gigabytes), and their static nature limits benchmarkers to characteristics already observed in the repository.

PipBench will instead populate a PyPI mirror directory with artificial, automatically-generated packages. As such, PipBench will be a file system image generation tool. The images generated will include compressed package archives, each containing executable Python files (including install scripts) and other assets. Benchmarking tools for gener-

ating images must consider which file characteristics are important and should resemble real file system images; for example, Impressions [3] reproduces aged file systems, and SDGen reproduces compressibility patterns [7].

Files in each package of the image will vary in size and quantity. Their characteristics will be specifiable via configurable distributions, but choosing configurations that meaningfully resemble real packages will be challenging. Thus, we will make it possible to templatize real packages to emulate their directory structure, file sizes, and dependencies on other packages. PipBench will then be able to create a realistic testing environment by populating the image with new, artificial packages stamped from such templates.

We also plan to inform the design of PipBench by studying how packages are used in practice. In particular, we hope to understand the dependency structure between packages by analyzing the PyPI repository and package popularity by analyzing Python projects on GitHub.

6. Conclusion

"Speed wins in the marketplace."
– Adrian Cockcroft [5]

Services that can be rapidly designed, implemented, and deployed have an important competitive advantage over their slower counterparts. Many different software engineering methodologies ultimately suggest the same approach to achieving higher velocity: *to be fast, one must be small*. The lean-startup approach dictates that in order to quickly iterate on product ideas, one must build and evaluate a series of minimum-viable products [15], the Agile development methodology dictates that in order to efficiently develop software, one must deliver minimal improvements frequently [16], and the microservice model dictates that in order to deploy software rapidly, one must decompose applications into minimal, easily deployable services [12].

This paper explores a dilemma in which minimizing work in one area creates more in another. In particular, engineers often minimize development time by leveraging existing libraries instead of writing everything from scratch. Unfortunately, this dependence can make a software bundle large and slow to deploy; in this case, increasing the development velocity decreases deployment performance.

We explore this problem in the context of Python packages used by OpenLambda handlers. We propose building support for the PyPI repository directly into the OpenLambda platform. Making the system aware of packages will make it possible to share libraries between different handlers. Containers have already enabled the sharing of kernel-space subsystems between containers (*e.g.*, file systems and networking stacks); sharing of more user-space resources is the next logical step towards greater efficiency. We believe Pipsqueak will afford developers the convenience of large libraries without sacrificing the performance of fast, lean Lambda handlers.

References

[1] AWS Developer Forums: Java Lambda Inappropriate for Quick Calls? https://forums.aws.amazon.com/thread.jspa?messageID=679050, July 2015.

[2] Pip Installs Packages. https://pip.pypa.io/en/stable/, May 2016.

[3] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. *ACM Transactions on Storage*, 5(4), November 2009.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[5] Adrian Cockcroft. Microservices Workshop: Why, what, and how to get there. https://www.slideshare.net/adriancockcroft/microservices-workshop-all-topics-deck-2016, 2016.

[6] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[7] Raúl Gracia-Tinedo, Danny Harnik, Dalit Naor, Dmitry Sotnikov, Sivan Toledo, and Aviad Zuck. SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 317–330, Santa Clara, CA, 2015. USENIX Association.

[8] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[9] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association.

[10] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[11] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 293–304, New York, NY, USA, 2002. ACM.

[12] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, February 2015.

[13] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 205–216, San Jose, California, October 1998.

[14] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[15] Eric Ries. *The Lean Startup*. Crown Business, September 2011.

[16] James Shore. *The Art of Agile Development*. O'Reilly Media, October 2007.

[17] Nikolai Philipp Tschacher. Typosquatting in Programming Language Package Managers, 2016.