λ Bureau of Investigation

Juan Esteller
esteller@college.harvard.edu

William Qian
wqian@g.harvard.edu

Abstract

Serverless computing is an emerging computing paradigm where users can focus solely on the business logic of their applications without needing to invest in system administration. Cloud providers like Amazon AWS [4] provides users with the platform, services, and tools to build, deploy, and maintain their applications. Still missing, however, are comprehensive tools for debugging applications on serverless platforms like AWS Lambda [5]. In this work, we identify core debugging features that are unavailable to Lambda users and present two technical innovations to improve the debugging experience. The command line step debugger allows a user to step debug a single Lambda instance, and the investigation toolkit provides the user with the ability to extract details from, interrupt programs on, and interact directly with individual Lambda instances. These tools fill a previously-unfilled niche in debugging serverless functions on AWS Lambda.

1 Cloud Computing at Present

One view of modern cloud computing models is that of a spectrum, from serverless computing at one end to serverful computing at the other, with cluster management tools like Borg and Kubernetes in the middle [11]. Cluster management tools and serverful computing have mature ecosystems of debugging tools, such as interactive debugging tools like the GNU Project Debugger (GDB) [10], much of which is currently absent in serverless computing. In this work, we focus on the popular Amazon Web Services (AWS) [4] and its serverless computing platform, AWS Lambda [5]. We identify interactive debugging as a crticially-missing feature for AWS Lambda and present two tools that tackle this absence. One tool allows for step debugging in AWS Lambda and the other allows for individualized control of an AWS Lambda instance.

1.1 Serverful Computing

In traditional serverful cloud computing, such as AWS Elastic Compute Cloud (EC2) [3], a user pays the cloud provider for a virtual machine (VM). This VM comes with compute, memory, and network resources, including a public IP address, which allows the VM to serve a public web application. Additionally, the user has direct access to the VM, which allows the user to use the same debugging tools that are available on the user's personal machines. When an application encounters an anomaly, the user can remotely access the VM as a highly privileged user and leverage a familiar, feature-rich set of debugging tools to identify and solve the problem.

However, deploying today's complex modern systems on serverful computing models can lead to many challenges. Servers that require orchestration with other servers, such as the task of load balancing, can only present the user with a local, narrow view of the system. As the system scales up, the user may need to examine more VMs at once to find and fix anomalies. Without advanced logging and tracing infrastructure in place, such as Canopy [12], the user will quickly be overwhelmed by the number of VMs they need to access and the intellectual overhead of correlating logs and data across many machines.

Cluster management systems tackle this problem of information overload in large serverful computing deployments with the concept of a job. By abstracting away the hardware details of a running application, jobs provide the user with a contextual, high-level view of the underlying system. While debugging, the user can rely on the cluster management system to understand which machines to target for direct access, thus reducing the cognitive burden of deducing the context themself.

Borg [1] is a cluster management systems that deploy jobs on long-lived servers. A featureful UI allows a user to submit, update, monitor, and kill jobs without needing to manually access each machine, but does not prevent such access. If desired, a user can directly access a a server through a public-facing IP address, which can be retrieved from Borg. This allows the user to debug a server as if in a serverful computing model. Thus, a Borg deployment provides valuable contextual information for the user while maintaining their visibility into and control over individual servers.

Canopy [12] is Facebook's end-to-end tracing system, which propagates metadata about each request to enable cheaper and faster analytics. Facebook's deployment of Canopy has allowed their developers to focus more on their projects, without worrying about how to implement and analyze their own metrics. Furthermore, by providing a common set of interfaces for use in any product, Canopy reduces the cognitive load on developers by only requiring them to learn a single general tracing library, rather than one library per product.

AWS Lambda [5] is AWS's serverless computing platform, and presents a different model of computing from serverful computing and cluster management systems. Unlike with EC2, a user cannot directly access an individual Lambda, which restricts the debugging tools available for investigating anomalies. Unlike Borg, there is no way to monitor in real time how each machine in the deployment is behaving [2], which clouds the user's visibility into the system. Additionally, Lambdas run on short-lived allocations, unlike the long-running servers in serverful computing and cluster management systems. Without these controls and visibility options, the user must find alternative ways to debug anomalies on Lambdas.

One approach is to use the AWS Lambda Editor, a unified web IDE that allows the user to quickly iterate on their Lambda. The user can define the test case's input parameters and immediately view the result of a run. Code changes can be immediately rerun to observe the effects. The editor does have a size limit, however, and will not serve projects greater than 3MB in size [6]. Additionally, the console editor does not provide access to any command line tools, which prevents the user from knowing what dependencies already exist on the system and from installing missing dependencies through tools like pip.

Another approach is to use the AWS Serverless Application Mode (SAM) [7]. SAM allows the user to locally emulate the environment in which Lambdas run. This gives the user full control of the runtime environment, and enables the same suite of debugging tools as serverful computing and cluster management systems. The main detraction of SAM is that not everything in the cloud environment can be easily emulated. Anomalies arising from heterongeneous hardware, variances in operating systems, and other environment differences will only be discovered on the actual cloud environment. In the absence of similarly-powerful debugging tools, the user will be ill-equipped for addressing these kinds of bugs when they occur on a live deployment.

AWS CloudWatch [8] provides logs as another way to debug Lambdas. Lambdas can write unstructured data to logs, which CloudWatch will automatically capture and store. When an error occurs, the user can inspect the logs for clues about the error. CloudWatch also provides higher-level analytics, such as error counts and success rates. Since CloudWatch is a tool provided directly by AWS, the user does not have to worry about log rotation, durability, or storage capacity, making CloudWatch an easy-to-setup tool for gathering clues about Lambdas. Since CloudWatch is a logging tool, though, it has two major limitations. First, logs can only provide information about what has already occurred – it cannot be used to speculate on whether a change will fix the problem. Second, CloudWatch can only provide logs, and cannot provide the user with any deeper level of access for debugging.

AWS X-Ray [9] can enrich the user's understanding of the situation by providing end-to-end traces for Lambdas. This is especially useful for when the Lambda runs as part of a larger stack, such as serving web content. While X-Ray does not operate at Canopy's scale, it can still provide useful information about which events are triggered, how often, and so forth. Like with CloudWatch, however, X-Ray can only provide information about Lambdas that have already been run, and cannot provide any finer-grained information about what the state of the program is, or whether a change will fix the problem.

AWS's suite of tools can help the user gather clues and understand the situations behind many failures in Lambdas, but still lack the ability to debug a Lambda as it encounters the bug. Live debugging can be useful in many situations, such as killing a Lambda before it corrupts the system's state or being able to step through a rare bug. We present a command line debugger and an investigation toolkit to improve the debugging experience.

2 Design

We propose two approaches for simplifying the debugging process of Lambdas. The first approach enables step debugging for a Lambda. The second approach uses a toolkit that enables fine-grained monitoring of and investigation into the state of individual Lambdas in a job.

2.1 Step Debugging

A debugger, such as the GNU Project Debugger (GDB) [10], allows a user to inspect the internals of a running program. A step debugger extends this by allowing the user to inspect the state of the program one statement or instruction at a time. Using step debugging, a user can pinpoint when an error occurs, what the state is, and what code path led up to the error.

AWS does not natively provide support for step debugging Lambdas; therefore, we must remotely connect a step debugger to a Lambda. One model for a remote debugger initiates a connection with the target binary, which in turn has an open port that listens for such connections. This model does not work for step debugging AWS Lambdas, however, because Lambdas are not addressable and cannot accept incoming connections. This means that there is no way for a user to initiate a connection with the Lambda from outside or inside the AWS network.

Our solution to this problem is to reverse the relation between the debugger and the Lambda. Rather than have the debugger connect to the Lambda, we have the Lambda connect to the debugger instead. Since AWS does not restrict outbound connections from Lambdas, this approach avoids the network restrictions that AWS has in place.

Having the Lambda connect to the debugger does have other restrictions. Since the debugger is now the server, the Lambda now has to address the debugger. This means that the user must provide the debugger's address and port *a priori*. Additionally, the user must ensure that the listening port is open and listened on when the Lambda is invoked.

2.2 Investigation Toolkit

A step debugger works well for reliably-reproducible bugs in a single Lambda. When a bug cannot be reliably reproduced, or when a job invokes many concurrently-running Lambdas, however, this approach becomes infeasible. In both cases, the user may have to sift through hundreds or thousands of debugging sessions, which is exhausting and impractical. Our investigation toolkit tackles this challenge by monitoring Lambda invocations for abnormalities and reporting just the abnormalities to the user. While it lacks the fine-grained control of a step debugger, our toolkit allows the user to understand and contexualize the environment in which the abnormality occurred.

The toolkit comprises two components: the *monitor* and the *dashboard*. The monitor runs on the Lambda and the dashboard presents a UI to the user for interacting with each Lambda's monitor. Additionally, we maintain the spirit of working in the serverless world and avoid using a central server for coordination and communication.

2.2.1 Monitor

The monitor runs on each Lambda instance to gather information about the Lambda and communicate with the user through the dashboard. We focused on the monitor's usability and versatility in our design.

Usability Since the monitor runs as part of the Lambda function, the user must actively incorporate the monitor into their function. Because this is a tool to assist the user, we must ensure that incorporating the monitor is as painless as possible. To achieve this, we design the monitor as a wrapper that encapsulates the function code. This allows the user to use the monitor library without changing any logic to the function code being monitored.

Versatility We have three required features for the monitor:

1. *Extracting* details about the runtime environment and communicating them to the user,
2. *Interrupting* the program at the user's will, such as allowing ad-hoc killing of Lambdas, and
3. *Interacting* with the user in a stepping manner, such as interposing before every network I/O request.

These features impose several requirements on the monitor. Extraction requires access to the Lambda's runtime information. Interruption and interaction occur during the function's execution, thus requiring concurrency. All three features require two-way communication between the monitor and the dashboard. We address these requirements as detailed below.

Extraction occurs at the beginning of a monitored Lambda invocation. To access the Lambda's runtime information, the monitor searches in two areas: the context and the language. The context is a parameter that AWS provides when the Lambda is invoked. This mostly contains job information, such as the Lambda's name, memory limit, and deadline. To retrieve information about the machine on which the Lambda is running, we rely on libraries in the language runtime. This can

provide information about the machine's memory, uptime, and MAC address. In order to uniquely identify a Lambda invocation, we combine the machine's MAC address, the function's name, and other parameters of the invocation. The monitor then sends the identifier and extracted information off to a data store for later retrieval. Upon a successful execution of the wrapped function, the record is deleted from the data store.

Interruption and interaction occur concurrently with the execution of the Lambda function's body. We therefore implement both features in an asynchronous monitor for the duration of the execution. To receive messages from the user, the asynchronous monitor polls from an AWS SQS queue. Each Lambda invocation has a corresponding SQS queue, which eliminates the need for filtering messages. Thus, once a message has arrived, the monitor can immediately act on it. The action taken depends on the type and contents of the message. We currently support one interruption (killing) and one interaction (stepping) message.

Interactions also need to send messages to the dashboard. While we could implement this with another SQS queue, we instead use the aforementioned data store that contains information about the Lambda's runtime. In order to send a message to the dashboard, we update the record, and AWS sends the update to all subscribing dashboards. This implementation has a critical advantage over one using an SQS queue. When a message arrives in an SQS queue, the consumer must decide whether to pop it off the queue. Doing so would allow the consumer to access the next message, but other consumers will no longer be able to read that message. Not doing so means that unless another consumer pops off the message, all of the consumers will be stuck reading just the first message. This makes an SQS queue inherently incompatible with the ad-hoc subscription model that we envision for the dashboard. The data store, however does provide the ability to publish a message to arbitrarily many subscribers by means of AWS AppSync.

2.2.2 Dashboard

As the user-facing part of this project, the dashboard is designed to present a clean, concise view of the information we have on each Lambda and a simple interface for interacting with each monitor. In this design, the dashboard performs three roles:

1. *Presenting* users with information about,
2. *Subscribing* to and displaying messages from, and
3. *Sending* user-initiated responses to each Lambda.

Given that the frontend web stack has many AWS APIs and is a generally good choice for user interfaces, we built our dashboard as a web application.

To display information about each Lambda to the user, the dashboard subscribes to the AWS AppSync API of interest. When a monitored Lambda is invoked, the insertion of a record into the associated data store triggers an event that sends the record to each subscribing dashboard. The dashboard can then parse the structured record for information and present it to the user. Upon deletion of a record in the data store, AWS AppSync sends a corresponding deletion event to each subscribing dashboard. Each dashbard then deletes the associated record from the UI. If a record persists beyond the expected timeout of Lambda, the dashboard alerts the user of the anomalous Lambda. A dashboard can also receive an event when a monitor updates the record in the data store.

To send a signal to a given Lambda, the dashboard constructs a message and pushes it to the corresponding SQS queue. This queue is identified by a URI that is provided as part of the record in the data store. The dashboard exposes this feature to the user as a set of buttons. Pushing a button will send the corresponding signal by enqueueing the message to the SQS queue. For example, to send a kill signal to a long-running Lambda, the user can click the kill button, which then sends the kill message to the monitor. To ensure that each message is sent no more than once, we use SQS's API to determine and filter duplicate messages.

3 Implementation

We implement both the step debugger and investigation toolkit's monitor library only in Python due to Python's flexibility and ease of use for a prototype implementation. Thus, our implementation currently only supports Lambdas written in Python. The investigation toolkit's dashboard is written in NodeJS and ReactJS.

3.1 Command Line Step Debugger

The default Python debugger is Python's native pdb [13] module, and can connect to any locally-running Python program. Rpdb wraps around pdb and enables remote debugging through an open socket connection with the target program. Since rpdb relies on the ability to address the remote program, which as mentioned in § 2.1 is absent on AWS Lambdas, we created crpdb to support remote debugging of Lambads running in Python.

```
 rpdb/__init__.py | 8 +++-----
 1 file changed, 3 insertions(+), 5 deletions(-)

diff --git a/rpdb/__init__.py b/rpdb/__init__.py
index c79c622..2132efd 100644
--- a/rpdb/__init__.py
+++ b/rpdb/__init__.py
@@ -43,18 +43,16 @@ class Rpdb(pdb.Pdb):
         # Open a 'reusable' socket to let the webapp reload on the same port
         self.skt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
         self.skt.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
-        self.skt.bind((addr, port))
-        self.skt.listen(1)
+        self.skt.connect((addr, port))

         # Writes to stdout are forbidden in mod_wsgi environments
         try:
-            sys.stderr.write("pdb is running on %s:%d\n"
+            sys.stderr.write("pdb is connecting to %s:%d\n"
                              % self.skt.getsockname())
         except IOError:
             pass

-        (clientsocket, address) = self.skt.accept()
-        handle = clientsocket.makefile('rw')
+        handle = self.skt.makefile('rw')
         pdb.Pdb.__init__(self, completekey='tab',
                          stdin=FileObjectWrapper(handle, self.old_stdin),
                          stdout=FileObjectWrapper(handle, self.old_stdin))
```

Figure 1. Code difference between crpdb and rpdb. Rpbd's server socket code is changed to client socket code.

Crpdb extends rpdb by flipping the client-server relation between the debugger and the target program, as shown in Figure 1.

3.2 Inspection Toolkit

3.2.1 Monitor

The monitor is implemented as a decorator class, which allows it to be easily added to any existing Lambda handler. The decorator runs in three stages:

1. Pre-execution
2. Execution
3. Post-execution

In the pre-execution phase, the monitor collects runtime information, sets up the Lambda's SQS queue, inserts the Lambda's record into the data store, sets up interpositioning in the AWS Python API, and spins up the monitor thread. To collect runtime information, we use the os, sys, and uuid Python libraries. We then invoke a GraphQL mutation to update DynamoDB with the Lambda's record, including the URL of the Lambda's SQS queue.

In the execution phase, the main thread invokes the original Lambda handler, and the monitor runs in a concurrent subthread as an instance of the LambdaMonitor class, a subclass of the Thread class. Messages are received in a loop by the LambdaMonitor, which polls from the SQS queue. When the function successfully completes, the main thread flips a flag, which tells the LambdaMonitor to break out of its polling loop and join back with the main thread. To ensure that the LambdaMonitor regularly checks this flag, we poll from the SQS Queue with a bounded wait time. Once we have received a message, the LambdaMonitor acts based on the encoded signal:

- **KILL** – indicates that the Lambda should be terminated.
- **PROCEED** – step over the current interposition.

The implementations of termination and interposition will be described below.

In the post-execution phase, the function has already successfully completed its execution and we can safely perform cleanup. The main thread signals for the LambdaMonitor to join, the SQS queue is deleted, and the Lambda's record is deleted with another GraphQL mutation. This triggers an event from AppSync to subscribing dashboards, which subsequently remove the Lambda's record from the dashboard UI. If the function encountered an error during the execution phase, the program will immediately terminate and cleanup will not occur; this allows the dashboard to identify anomalies when a Lambda's record has outlived the Lambda it represents.

3.2.2 Dashboard

We build our dashboard as a NodeJS + ReactJS web application. We choose ReactJS because it is one of the few frontend libraries with AWS AppSync API support, which we need for subscribing to Lambda record changes. We choose NodeJS for its compatibility with ReactJS and the AWS Amplify library, which generates the GraphQL schema, queries, and mutations that we need. The dashboard UI is implemented as a table, with rows representing Lambdas. Each row contains the runtime information about the Lambda, as well as actions for the user to take, in the form of the proceed, kill, and clear buttons, as shown in Figure 2. The proceed button only appears when the Lambda has been interposed. The kill and clear buttons are available at all times, and kill the function and clear the record from the data store, respectively.

3.2.3 Termination

We implement termination with the help of the Python threading library. This form of interruption works by interrupting the main thread and killing the process. In Python3, this is a call to `_threading.interrupt_main()`, and in Python2, this is a call to `threading.interrupt_main()`.

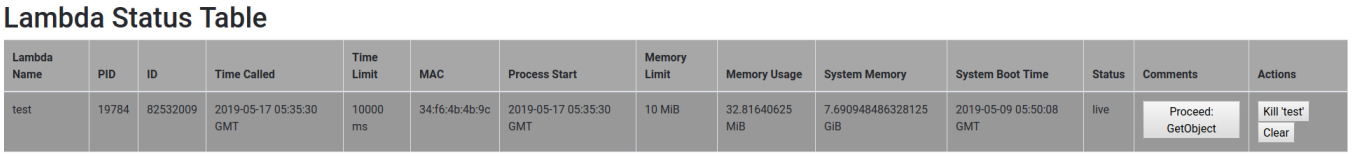

Figure 2. Dashboard with one running Lambda.

Due to the incompatible version differences, the LambdaMonitor dynamically checks its version when terminating to determine which implementation to execute. Once the interruption is made, the Lambda fails ungracefully.

3.2.4 Interposition

We currently support interposition on any AWS API call. By default, we interpose on all GetObject calls in the S3 API. When the function hits an interposed API call, the API call publishes its interposed state and waits on an Event that is triggered only when the LambdaMonitor receives a PROCEED signal from the user.

Like with the overall monitor design, we avoid the need for complicated code changes that the user might have to implement. Instead, we wrap the API creator `botocore.client.ClientCreator._create_api_method` to inject our interposition code into the AWS API. Since we are using Python, this trick allowed us to dynamically redirect all AWS API calls in the function to our wrapped version, without making any source code changes to the AWS API itself. We believe, however, that this implementation would not extend well to other languages, such as C++ and Java.

4 Evaluation

We evaluate only our investigation toolkit, as the command line step debugger is functionally the same as rpdb. We evaluate the toolkit on two use cases, an S3 retrieval and a PyWren deployment. Our evaluation takes into account three metrics:

1. The cost of setting up and using the toolkit,
2. The usefulness of the dashboard for identifying and understanding anomalies, and
3. The performance overhead for using the toolkit.

The first two metrics are qualitative, and the third metric is quantitative.

4.1 S3 Retrieval

The S3 retrieval test makes a call to S3 to retrieve an object before exiting. We include this test in our library as `test.py`, and invoking the test with `python3 test.py` will start one run of the test. Multiple concurrent runs can be achieved with multiple invocations.

4.2 PyWren

The PyWren deployment is in `script.py`, and can be run with `python3 script.py`. The PyWren workload we provide is a short-lived number squarer. Unlike the S3 retrieval test, we cannot directly decorate the Lambda function handler without making changes to PyWren. Instead, we wrap the PyWren handler inside our own handler, on which we apply the monitor library decorator. This handler wrapped can be found in `index.py`. To run the test, our run script first deploys PyWren normally. We then download PyWren function, add libraries that we need, and reupload the code with our changes. We also change the Lambda's entry point to point to our wrapping handler, instead of PyWren's handler.

4.3 Setup Cost

We designed the monitor library to be a minimal-cost library for users. The S3 Retrieval test demonstrates the minimal work needed to use the monitor library: importing the library and adding the decorator to the function handler. To use the decorator, the user must explicitly specify the AppSync API URL and API key, which the user can easily retrieve from the AWS dashboard. In a situation where the user does not easily have direct access to the source code of the function handler, like in PyWren, the setup cost rises. In addition to using the decorator, the user must also wrap the target handler and redeploy the Lambda. We have automated this process for PyWren, but customizing the redeployment for every new framework can be difficult.

Another half of the setup cost is setting up the AWS services. In order for the investigation toolkit to work, the user must set up AppSync, DynamoDB, the GraphQL API, and either Cognito or IAM. Although we provide the schema for setting up GraphQL, this process can still be difficult. Additionally, AppSync keys can expire,

which means that the user may need to frequently renew their keys and update them in the decorator, which requires another round of redeployment.

Overall, we believe that we have made the basic use case for the investigation reasonably frictionless. Some future work in this area can seek to automate the AWS setup procedure, which can relieve the user of infrastructural burdens. Further automation can automatically wrap function handlers and redeploy the function.

4.4 Dashboard

Figure 3 shows the dashboard during a PyWren test, with interpositioning and record clearing disabled. We can see that the dashboard provides a concise summary of many details about each individual Lambda instance. Unique identifiers allow the user to distinguish the Lambdas from each other and cross-reference with other services, like CloudWatch. Furthermore, the dashboard dynamically updates itself with the latest information, so the user can passively monitor their Lambdas. The combination of a friendlier UI and visual presentation helps the user to better understand how their Lambdas are running.

For interpositioning, an interactive feature, the user is presented with buttons to use. With the default interpositioning configuration, the LambdaMonitor will pause the function at every S3 GetObject call, and reflect this in the dashboard. Figure 2 shows the dashboard for one run of this test case. The interposition is reflected in the Comments column by a button that prompts the user to send a PROCEED signal, as well as the AWS API call that triggered the interposition.

Overall, the dashboard does present to the user a broad view of their Lambdas, a feature that does not currently exist on AWS. We still suffer, however, from information overload, rapid fluctuations of rows appearing and disappearing, and other UX concerns. Future work in this area could include better ways to group Lambdas by job and a more customizable dashboard.

4.5 Performance Overhead

We used lightweight statistical counters to report the time cost of each step in our monitor library. Based on sample runs of the PyWren test with 10 Lambdas per job, the overhead for the pre-execution phase is 2 ± 1 seconds, and for the post-execution phase is 3 ± 1 seconds. These costs are fixed, and therefore are more significant for short-running Lambdas than for long-running ones. We also only measured these costs for warm Lambdas, as we expect that the toolkit's overhead will be proportionally less in a cold start, where other costs, such as spinning up the VM, can dominate the performance profile. Overall, we believe that our toolkit's performance overhead is generally low enough for most applications, though future work can look into better understanding the performance profile and optimizing it for better performance.

5 Discussion

5.1 Monitor Usability and Versatility

In § 2.2.1, we presented usability and versatility as core components of our monitor library design. We briefly return to these topics. Based on our experience of incorporating the monitor library in our test cases, we have found that the decorator design is a convenient and concise way to incorporate the monitor without relying much on the user. Since we hold to the tenet of not expecting the user to modify their code beyond including the decorator, the monitor library presents a very low bar to entry. We have also demonstrated that the monitor library can extract details, interrupt functions, and interact with the user, which satisfies our desire for versatility. Though our implementation still has many areas for improvement, it does succeed in demonstrating the feasibility of our main desired features.

5.2 Future Work

Though our implementations are imperfect, they both fill a niche for debugging Lambdas that is presently unoccupied. Future work in this direction can look at more automation and better presentation of details in the system, or integrating these contributions with other debugging tools to facilitate easier debugging for users.

One direction in integrating tools is integrating the step debugger with the investigation toolkit. For example, if the LambdaMonitor can interpose between when the function encounters an error and when it exits, then it can provide the user with option of connecting that Lambda to the step debugger, allowing the user to step debug fewer Lambdas.

One possibility for more automation is a setup script that uses AWS command line tools to walk the user through setting up AppSync, DynamoDB, the GraphQL API, and identity management. These are all distinct components of the AWS ecosystem, and their individual quirks can add up to be overwhelming for even experienced users. A setup script can help relieve the

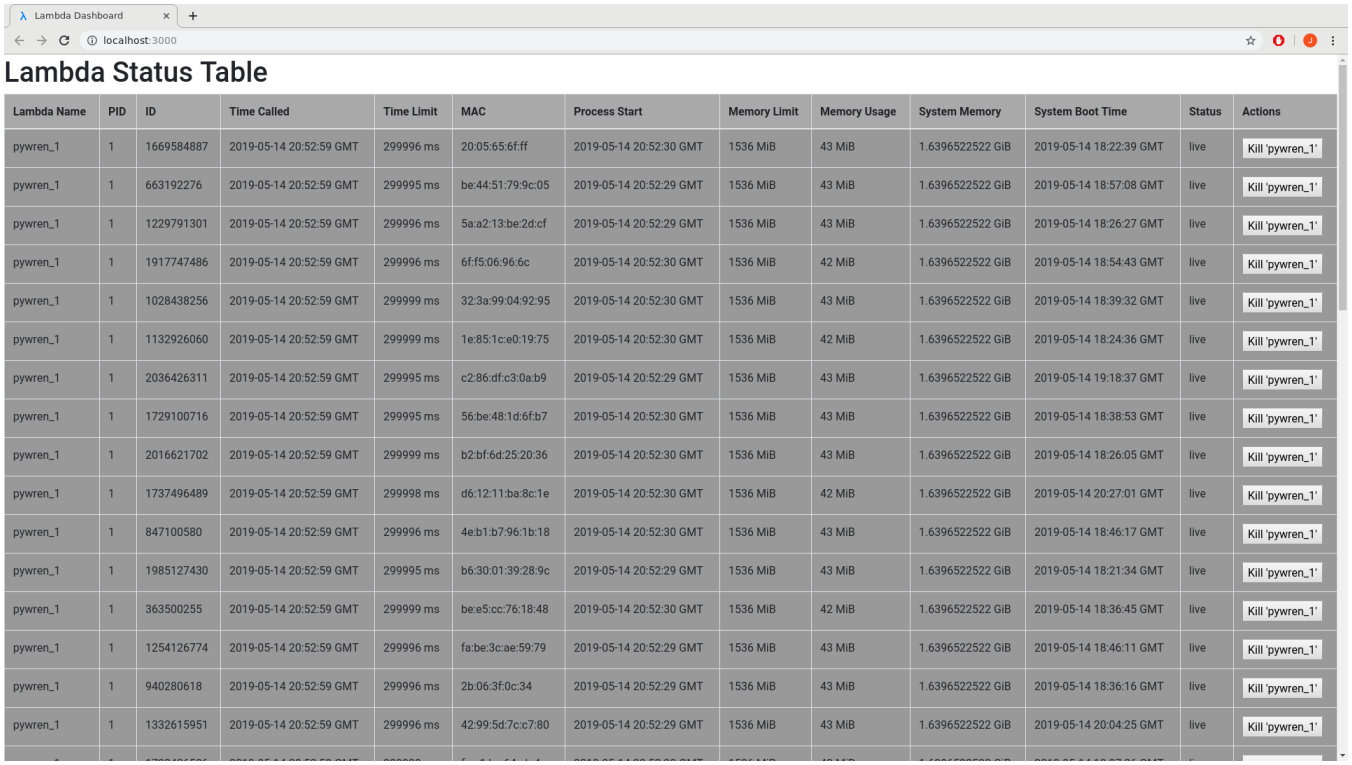

Figure 3. A screenshot of the dashboard during a PyWren run.

burden on the user, as well as reduce the likelihood of a fatal misconfiguration.

Another direction for automation is in automating the monitor library inclusion for complex deployments like PyWren. While we currently have a way to automate the PyWren deployment, the script is complex, and we expect that it will be similarly complicated for other similar deployments.

Presentation-wise, a more customizable UI with more options for interacting with the Lambdas can help improve the versatility of the toolkit. Since we have already implemented an interruption and an interaction, future work can use these implementations as templates for other interruptions and interactions.

6 Conclusions

Our project demonstrates inroads in improving the debugging experience for AWS Lambda users. We prescribe a simple change to `rpdb` that allows for step debugging in Python. We also built the Lambda Investigation Toolkit for transparency into each Lambda running in a job. The toolkit provides introspection into the specification and state of each Lambda, presented to the user through a dashboard UI. Through the dashboard, users can also send signals to an individual Lambda, such as KILLing a looping Lambda or telling it to PROCEED past an interposition.

References

[1] Abhishek Verma, Luis Pedrosa, et al. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*.

[2] Amazon Web Services. 2016. Can we check how many concurrent lambda currently run? https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutesuws.amazon.com/thread.jspa?threadID=222549

[3] Amazon Web Services. 2019. Amazon EC2. https://aws.amazon.com/ec2/

[4] Amazon Web Services. 2019. Amazon Web Services (AWS). https://aws.amazon.com/

[5] Amazon Web Services. 2019. AWS Lambda – Serverless Compute. https://aws.amazon.com/lambda/

[6] Amazon Web Services. 2019. AWS Lambda Limits – AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/limits.html

[7] Amazon Web Services. 2019. AWS Serverless Application Model. https://aws.amazon.com/serverless/sam/

[8] Amazon Web Services. 2019. Using AWS CloudWatch. https://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions.html

[9] Amazon Web Services. 2019. Using AWS X-Ray. https://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html

[10] Free Software Foundation, Inc. 2019. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/

[11] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[12] Jonathan Kaldor, Jonathan Mace, et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Symposium on Operating Systems Principles*.

[13] Python Software Foundation. 2019. pdb— The Python Debugger. https://docs.python.org/3/library/pdb.html