Distributed Testing via Serverless Functions

Sanat Deshpande, Lionel Eisenberg, Jon Karyo, James Lubowsky
Johns Hopkins University

Abstract

When suites of unit tests are introduced into a large code repository, identifying the provenance of failures can be a daunting task. While tools such as git bisect already exist to determine the last failing commit in a repository, this approach comes with the limitation of *O(klog(n))* performance (where *k* is the runtime of a test, and *n* is the number of commits), and poor feedback as to exactly which tests failed in each commit. We introduce the proof-of-concept of a tool that tests all *n* commits in parallel to achieve *O(k)* performance, and provides intuitive visual feedback as to which tests specifically failed. Our workflow begins as the programmer specifies the past *n* commits to test. The repository in question is cloned locally, and each of the *n* commits is checked out and uploaded to an AWS S3 bucket, whereupon an AWS Lambda function containing testing code executes upon the commit. Results are written to another S3 bucket and fetched to display to the user. Comparing our performance of the tool with git bisect, we notice that the rate-limiting step of our tool completes faster than the entirety of git bisect as the number of commits increases and as the input file size increases. For tasks that are computationally complex, our tool provides a swift and informative analysis of failures which outperforms other current methods.

1. Introduction/Motivation

Github is an immensely popular version control tool across the board, from small side projects to enterprise level endeavors. As standard practice on such projects dictates having thorough tests to ensure the functionality of current code [1], and that of any new changes introduced into the codebase, problems identifying the exact failures can arise as the project scales. Take, for example, a suite of tests that are added along the development of product. If several of these tests start indicating failures, it can be difficult to pinpoint the exact commit along the way where these tests would have begun failing.

Tools already exist to address searching through commits quickly to identify failure points. Git bisect [7] is one such example that finds failing commits with *O(klog(n))* performance. This command line tool allows the user to define what constitutes a working commit and uses this criteria to determine which commits pass or fail the given test. Git bisect performs a binary search from a specified start and end point in the commit history to find the first one that fails. While this may seem fast on paper, the programmer must bear in mind that testing each of these commits could take a non-trivial amount of time, especially for large projects with extensive testing suites. Another shortfall comes in git bisect's limited feedback. The tool only assesses a commit as "passing" or "failing", with no information regarding exactly which tests pass or fail. As issues with code, especially in production, can cost companies large sums of money, we aim to explore even faster methods of identifying failing commits.

Our solution aims to capitalize on Amazon Web Services' Lambda functions [3]. These are one-off functions [4,5,6] that we can call on demand to run a suite of tests against a particular commit in our codebase. In fact, we can run one instance of our Lambda function for every commit of interest *in parallel* [7], which should identify the failing commit in *O(k)* time. Our tool further provides intuitive, visual feedback regarding exactly which tests in which commits failed.

It is our goal that in cases where finding a production-critical bug is of high monetary value, the speed and cost of our tool would outstrip that of conventional tools like git bisect.

2. Automated Lambda Design

We created an interactive tool in the form of a website in order to perform severless testing. The tool is composed of a very basic frontend that mandates that the user specify the link to their Github repository that they wish to conduct unit tests on and the number, *n*, of previous commits. After these variables are specified, the frontend makes sequential HTTP requests to our backend to clone the last *n* commits, zips each commit, names it in sequence, and uploads the *n* zip files to an Amazon S3 source bucket.

We created an Amazon Lambda that triggers once an object is uploaded into the source bucket. However, the actual logic of the Lambda needs to be configured by the user to accommodate for their specific needs regarding testing. Our Lambda always unzips the object and will then execute the appropriate specified test the user wishes to run on the source code located within the file. Once the testing is complete, the Lambda is programmed to upload the output file which states the success or failure of the testing job to an Amazon S3 target bucket.

While the Lambda is running, the tool continuously queries the S3 target bucket every *x* seconds to determine the size of the bucket. Once the bucket is of size *n*, all tests would have finished. At that point, the tool fetches all of the information regarding the status of each test and displays it to the user in an easily readable manner with an added field which indicates the duration of the Lambda execution for analysis in Section 3.2.

Finally once all the results have been acquired, the frontend queries the backend which deletes all the files from our two buckets to reinitialize the process.

3. Methodology

We compared the time to find a failing commit using our tool against git bisect run on commodity hardware. To accomplish this task, we first created a file to test. This file simply performed selection sort, bubble sort and insertion sort on a specified input and compared the output to that of the python defined sort function to assess correctness. After creating the file, we uploaded it to a Github repository and made gradual changes that would produce incorrect comparisons which were also pushed to the repository. The code that produced intentionally erroneous output was labeled as the code-breaking commit which we hoped to identify using git bisect and our tool. The point of failure was always the second to last commit to ensure that git bisect would always need to perform *O(log(n))* comparisons. This enabled us to analyze the asymptotic worst case scenario. We assessed the time of completion for integer arrays of sizes 1000, 5000 and 10000 along with the past 5, 10, 15, and 20 commits for each input size to determine the effectiveness of our tool compared to the effectiveness of git bisect.

3.1 Git Bisect Testing

We ran automated git-bisect 10 times on our GitHub repository for the varying input

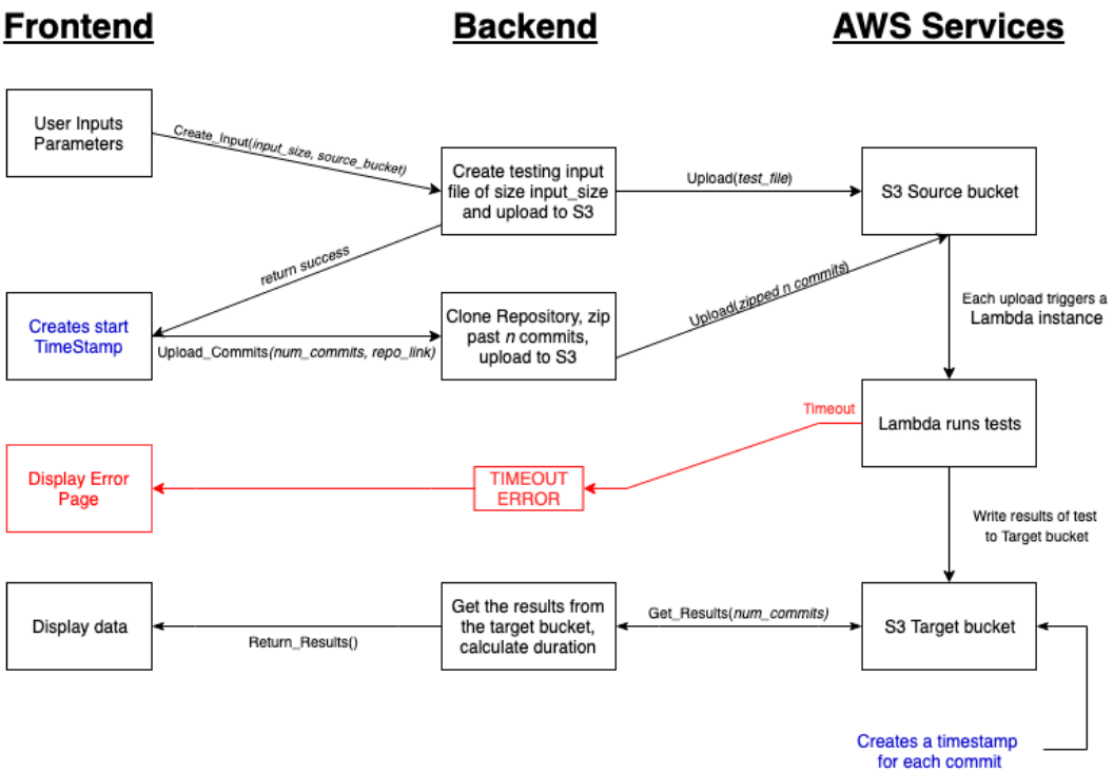

Figure 1: The workflow which includes benchmarking of our tool

sizes and looking at various numbers of commits as specified in the previous section on commodity hardware with 16 GB of memory and an Intel Core i7-6500U processor. We recorded the average time of completion for executing git bisect for each test case.

3.2 Automated Lambda Testing

3.2.1 Lambda Workflow

We used our automated Lambda testing tool on the same GitHub repository and assessed it's time to completion. Using our tool, we specified the link to the repository and how many commits we which to examine. We added an additional parameter so that the user can specify an integer, *m*, and the tool would automatically create and array of *m* random integers to be run with each commit to assess the correctness of the sorting function. Furthermore, we created Lambda functions that would call the commit of interest on the array of *m* integers to sort and compare that to the correctly sorted array performed by python's defined sort function. Our frontend queried the target bucket every 1.2 * *n* milliseconds.

3.2.1 Lambda Benchmarking

The two main parameters we were seeking to vary throughout our benchmarking were, *n*, the number of previous commits to test and, *m*, the input size of the file we would be testing on. For each combination of input size and number of commits, we ran our tool 10 times, recorded the average time it took for all Lambdas to complete and then averaged those numbers with each other for the 10 trials. Moreover, since our "limiting-time" is defined as the Lambda that takes the longest time to complete, we also took the longest duration for each iteration and averaged those over all the iterations. Finally, we took

standard deviation measures to be able to assess the variability in the Lambda completion time.

To get accurate times on how long our process took from start to finish, we first created a timestamp once the input file was created (as the input file upload time was negligible and out of the scope of the benchmarking needed). To get the end time of our process, we got the timestamp associated with the result files getting created from AWS S3 Metadata. Unfortunately these timestamps are only accurate to the 10th of the second, therefore all of our results have a 0.5 second error margin. Our workflow, which includes how we performed benchmarking, is summarized in Figure 1.

4. Results

We organized the collected data into two sets in order to fully analyze the speed of our tool in detecting when a code breaking bugs emerges compared to executing git bisect. Figure 2 depicts the average runtime of completing our tests using git bisect on commodity hardware, our tool, and the "limiting-time" of our tool as a function of the number of commits they are executed upon for a fixed integer array size for the three different cases (1000, 5000 and 10000 integers).

Figure 3 stratifies the data by the number of commits, so that we can visualize that relationship.

5. Discussion

After running our tool on a test repository with 20 planted commits, each either passing or intentionally failing our tests, and doing the same using git bisect, we can see how the performance of both tools varies. We see generally from Figure 2 and Figure 3 that our tool outperforms git bisect as the input file size increases, and that all tools tend to suffer

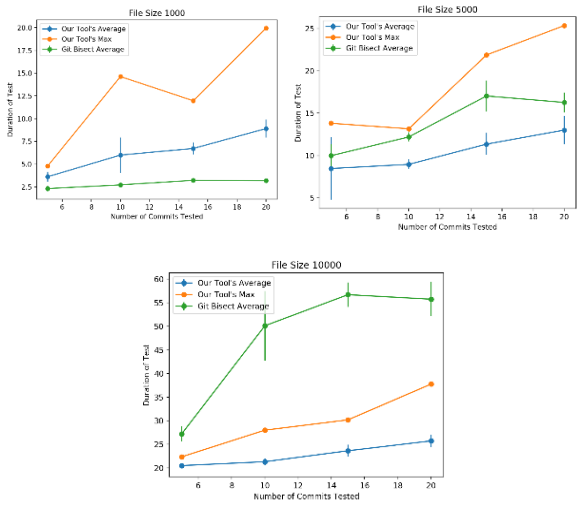

Figure 2: The time to complete testing as a function of the number of commits for a fixed file size using git bisect and our tool.

slightly in performance as the number of commits increases. Our relative improvement in performance as input file size increases follows the logic outlined in our motivation. Namely, the fact that Git Bisect operates in $O(klog(n))$ time means that it necessarily performs more sequential operations than our tool, which executes these operations all in parallel. Thus, while a test that takes longer to run means that our tool would also take longer to run, it in turn means that Git Bisect would take yet even more time to complete.

This also makes sense in the light of the functions we are testing, as they are sorting functions. The completion time of these tests correlates with an increase in input file size, as our tests are bottlenecked by our bubble sort function, whose performance slows quadratically. Extrapolating this performance comparison of git bisect and our tool, we can determine that the computation bottleneck is significantly smaller for our tool than for git bisect. Thus, as testing becomes more intensive and takes longer to complete, the performance of our tool in comparison to git bisect grows sharply.

The reason we see that all tools' performance degrades as the number of commits increases is different for our approach and for git bisect. Git bisect must

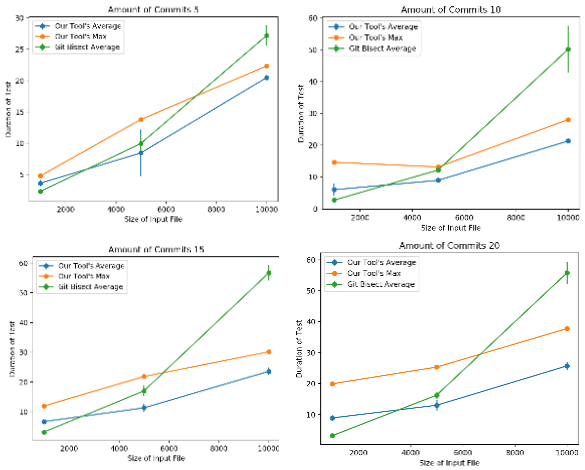

Figure 3: The time to complete testing function as a function of the input size for a fixed number of commits back using git bisect and our tool

execute *log(n)* tests in the worse case, so it is given that this number would grow as *n* increases. On the other hand, while our tool theoretically works in *O(k)* time, our particular implementation still uploads files one at a time to the bucket. While the upload time was trivial for us, this still introduces a slight bottleneck in our workflow. In addition, on occasion, our Lambda functions were slower to trigger on our file uploads, which introduced further, unpredictable delays. The chance of such a slow start increases as more commits are uploaded, explaining part of our performance decline with increased commits. As a last note, in practice, the most important metric for our tool would be the lambda function that takes the longest to run, depicted in orange in the graphs in Figure 2 and Figure 3. In every case, this rate-limiting operation still has the same performance trends relative to git bisect as the average case of our tool does. This means that in a real-world scenario, we would still expect to see improved performance over git bisect as the number of commits increases, and in particular, as the time each test takes to run increases.

6. Conclusion and Future Work

We tested the performance of our tool in scenarios with varying numbers of commits as well as varying input sizes for our tests – the latter being a good proxy for test completion time. We ran git bisect, the conventional tool for solving this problem, against the same battery of tests. Our data showed that while git bisect has better performance for low numbers of commits and low computational complexity, our tool, even in the worst case, outperforms git bisect as the number of commits and input file size increases. While this proof of concept illustrates not only the effectiveness, but also the efficiency of our tool over git bisect, there still remain several areas of further improvement.

Our current implementation lacks modularity. As it stands, we have a singular bucket with all the necessary permissions and read-accesses for the lambda functions configured. This restricts us to running our tool on only one repository at a time, and does not allow anyone to immediately download and start using our tool. Our next steps would thus certainly include modularizing our implementation to allow for the dynamic configuration of any number of buckets for the user as well as entry for credentials for interacting with private repositories.

An important benchmark to consider in future work would be a cost-benefit analysis. Since we aimed our tool to be used in the case where finding production-critical bugs was of monetary significance, we would need to measure performance against the cost of calling each lambda function. We noticed a performance improvement as we did some preliminary experimentation by increasing the memory limit of our lambda to be comparable to that of running git bisect locally, since our benchmarks were taking quite long to run. Formalizing this cost versus performance would be a critical next step in

bringing this project from proof-of-concept into production. Furthermore, being able to estimate the cost of running our tool would allow businesses to make critical decisions regarding the use of our tool in production-critical scenarios.

References

[1] "White Paper Why Bother to Unit Test?" *QA Systems The Quality Assurance Company*,www.qa-systems.com/fileadmin/user_upload/resources/White_Papers/Why_Bother_to_Unit_Test.pdf.

[2] Git-Bisect Documentation." *Git*, Git, git-scm.com/docs/git-bisect.

[3] Hendrickson, Scott, et al. "Serverless Computation with OpenLambda." *Usenix.org*, University of Wisconsin, www.usenix.org/system/files/conference/hotcloud16/hotcloud16_hendrickson.pdf.

[4] "Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions." *Future Generation Computer Systems*, North-Holland, 4 Nov. 2017, www.sciencedirect.com/science/article/pii/S0167739X1730047X.

[5] Stojanovi, Slobodan, and Slobodan -Stojanović "The Best Ways to Test Your Serverless Applications." *FreeCodeCamp.org*, FreeCodeCamp.org, 15 June 2018, medium.freecodecamp.org/the-best-ways-to-test-your-serverless-applications-40b88d6ee31e?gi=5cb72d309fd3.

[6] Wang, Liang, et al. "Peeking Behind the Curtains of Serverless Platforms." *Usenix.org*, UW-Madison and Ohio State University, www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf.

[7] *Lambda Architecture for Cost-Effective Batch and Speed Big Data Processing - IEEE Conference Publication*, ieeexplore.ieee.org/abstract/document/7364082.