Running Sikraken on EC2

Design Specification

Isaiah Andres

February 2026

Abstract 2
Introduction 2
Architecture Diagram - Elastic Compute Cloud (EC2) 3
- GitHub Actions - Pipeline Link 3
- Workflow 3
- S3 Bucket 4
- AWS Lambda 4

Architecture Diagram - Elastic Container Service (ECS) 5
- **User/Sikraken Developer** 5
- GitHub Actions - Pipeline Link 5
- Docker 5
  - Sikraken Container 5
  - Benchmarks Container 6
- Elastic Container Registry (ECR) 6
- Elastic Container Service (ECS) 6
- Simple Storage Service (S3) 7
- AWS Lambda 7

Architecture Diagram - AWS Batch 7
- Terraform 7
- GitHub Actions - Pipeline Link 7
- Docker 8
- AWS Batch 8
- Elastic Container Registry (ECR) 9
- Simple Storage Service (S3) 9
- AWS Lambda 9

References 9
Appendix 18
- Appendix 1: Lambda Script Sequence Diagram 18
- Appendix 1.1: Lambda Script Code 18
- Appendix 2: EC2 Instance Script Sequence Diagram 21
- Appendix 2.1: EC2 Instance Script Code 22
- Appendix 3: Filepath Processor Python Script 25
- Appendix 4: Lambda Script Sequence Diagram 28
- Appendix 4.1: Lambda Script Code 28
- Appendix 5: Run ECS Task Script Sequence Diagram 34
- Appendix 5.1: Run ECS Task Script 34
- Appendix 6: ECS Container Script Sequence Diagram 36
- Appendix 6.1: ECS Container Script 37
- Appendix 7: Terraform Diagram 51
  - AWS Resources 51
    - sikraken-batch.tf 51
    - sikraken-ecr.tf 52
    - sikraken-lambda.tf 52
    - sikraken-s3-bucket.tf 52

Input/Output Variables...52
outputs.tf...52
variables.tf...52
Setup And Environment...52
networking-data.tf...53
user-data.tf...53
terraform.tf...53
main.tf...53
Permissions...53
iam-aws-roles.tf...53
iam-github-roles.tf...53
oidc.tf...53
Appendix 8: Batch Container Script - Sequence Diagram...53
Appendix 8.1 Batch Container Script...54
Appendix 9: Run Batch Script...0
Appendix 10: Lambda Script Sequence Diagram...0
Appendix 10.1: Lambda Script...0
Appendix 11: Container Results Summary Processor - Sequence Diagram...0
Appendix 11.1: Container Results Summary Processor Script...0
Appendix 12: Report Container Script - Sequence Diagram...0
Appendix 12.1: Report Container Script...0
Appendix 13: Sikraken Dockerfile...0
Appendix 14: Reports Dockerfile...0

Abstract

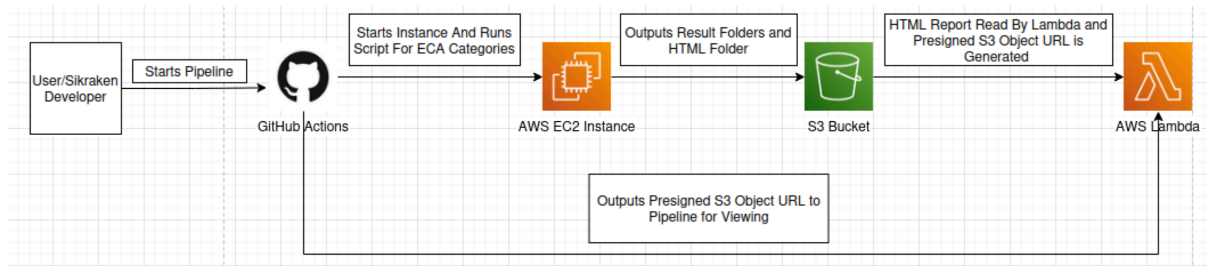
The purpose of this document is to model the current design for my final year project, which is the running of Sikraken on an EC2. This document will highlight the architecture diagrams for different implementations of the pipeline and explain the design decisions made, the pipeline's usage, the scripts used and their sequence diagrams.

Introduction

Sikraken is a test suite generator tool created by Dr Christophe Meudec [1]. It is run against Test-Comp's benchmark, where I may take 7300000 CPU seconds to run [2]. By running the tool along with the Test-Comp benchmark on multiple EC2 instances, or on a powerful EC2 instance, the time it takes to run the tool against the benchmarks may be drastically reduced.

This document will highlight the automation provided for testing Sikraken against the Test-Comp benchmarks, the architecture behind the each pipeline, and how to use it. The reason for the pipeline is so that Sikraken developers have a way to automatically view the results of any changes that they've made to the software, thus improving the efficiency of seeing the impact of the changes that they made.

Architecture Diagram - Elastic Compute Cloud (EC2)



The architecture diagram above illustrates the current flow of the pipeline.

User/Sikraken Developer

The user would typically be a Sikraken developer. The pipeline itself can be triggered manually or whenever a commit or a pull request is made, in order to quickly see the results of any changes made to the Sikraken repository. By the time the Lambda is finished running, the user will receive a presigned URL generated by the Lambda for the purpose of viewing the HTML report generated by one of the Sikraken scripts.

GitHub Actions - Pipeline Link

The platform that the pipeline runs on is GitHub Actions, as Sikraken is already on a GitHub repository. It is written in .yaml format as it's the only supported format and It serves as the core of the project, as it handles the automation and the running of the scripts within the EC2 instance through the use of AWS Systems Manager and is what invokes the Lambda. The presigned URL for the HTML report that's generated by the Lambda after the pipeline invokes it is also viewed from the pipeline.

Potentially sensitive information can also be used in the pipeline such as EC2 instance Ids without exposing them through the use of secret variables [74] and environment variables [119] can be used to configure parameters for the script run inside the EC2 instance.

Only those with write access can use the pipeline at the moment such as owners and contributors. The current pipeline has its jobs run by GitHub's runners, a machine that runs the jobs specified in the pipeline [118], allowing for any hardware concerns to be left to GitHub.

Workflow

The pipeline is set to be triggered if a commit has been pushed or a pull request has been accepted, and can also be set manually. The AWS credentials are then configured using OIDC with an IAM role associated with the AWS account for better security.

Afterwards, the EC2 instance, with its ID specified as a GitHub secrets variable for security, is started up and waited for, using the AWS CLI, before the script within the pipeline is run to avoid errors. The script itself is run once the EC2 instance is confirmed to be running using AWS Systems Manager (SSM), and the list-command-invocations SSM function is used to get the status and show the outputs made by the script, such as errors generated by Sikraken.

A Lambda function is then invoked once the script is running to return a presigned URL of the .html report stored in the S3 bucket before the EC2 instance is stopped at the end.

AWS EC2 Instance

The AWS EC2 instance used is the memory optimised version of an EC2 instance (r5.xlarge), as memory-optimised instances are better suited for workloads that process large data sets in memory and has four cores and 32GB of memory[13]. Given that Sikraken may be prone to using upwards of 12GB of memory on rare occasions [2], this model had the potential to be the best option.

The version of the EC2 instance being used is the Ubuntu version, the operating system Sikraken is being developed on, and runs a script to run Sikraken against the ECA benchmarks, process the .html file outputted and then upload the folders generated after a test run to the S3 bucket.

Currently, each pipeline run takes less than 7 minutes and given that the cost to run the EC2 instance per hour is $0.282, meaning that each pipeline run costs less than $0.0329094 [11]. Since there's a risk that code within Sikraken may not terminate itself despite being given a time budget, it's also necessary to implement a timer that would terminate the program in the Sikraken scripts.

(See Appendix 2 and 2.1 for the sequence diagram and script to generate a test run in the EC2 instance and upload the results)

S3 Bucket

The S3 bucket is used to hold all the folders generated by Sikraken following a test run against the ECA benchmarks. The AWS Lambda will be used to read the .html report generated by a script after a Sikraken test run that is uploaded to an S3 bucket. An S3 bucket is suitable for this task as it acts as cloud storage for folders generated after a test run and has additional metadata for easier processing within the Lambda [82] such as using the last modified.

It also doesn't matter what kind of EC2 instance is used, any of them can simply write to the same S3 bucket, allowing for change in EC2 instances to take place without any problem and is also cheaper at $0.023 per GB each month for the first 50 terabytes for the standard S3 [116] used in comparison to the EBS gp3 pricing at $0.088 per GB each month [23], both of which are models currently used for the project. A snapshot of the current EBS volume will also have to be taken and used in the event that a new EC2 instance is launched, and if there's a need to get the older test run data stored in the folders [117].

AWS Lambda

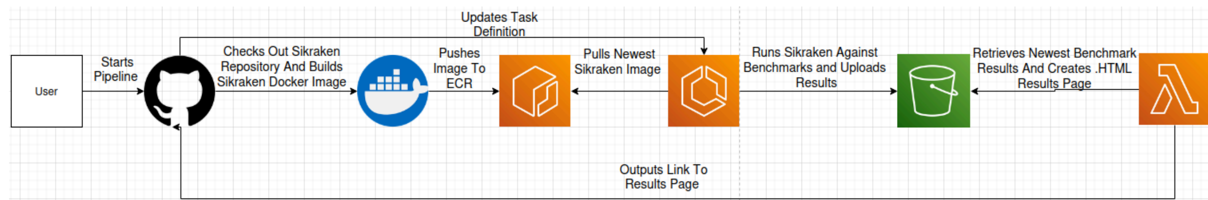
The AWS Lambda is used to read the .html file stored inside the most recent folder generated by Sikraken that is uploaded to the S3. Since only those with the right credentials can access the objects stored inside an S3, the Lambda generates a presigned URL for access to the .html report, which all users can access if they have the URL to the .html file [83].

A Lambda is used as it provides a simple way of deploying and running code on the cloud. It supports Python as a programming language, and the code deployed on a Lambda can easily be invoked through a pipeline, as well as have its output (the presigned URL) easily viewed [85].

If the code to read and output the .html file were to be executed in the EC2 instance or anywhere else, the AWS account's credentials would have to be passed within the code, adding more complexity just to access the boto3 SDK [120,121]. It's also already very cheap to invoke a Lambda, at 0.20 per 1 million requests [122], which may justify the convenience provided.

(See Appendix 1 for the sequence diagram and Appendix 1.1 for the code within the Lambda to find the code to generate a pre-signed URL)

Architecture Diagram - Elastic Container Service (ECS)



The above architecture diagram is an ECS implementation of the automation of running Sikraken against its benchmarks and would be faster as it allows for more virtual CPUs (vCPUs) to be used in comparison to the four cores that the current EC2 instance has.

User/Sikraken Developer

The user would typically be a Sikraken developer. Similarly to the EC2 pipeline, it can be triggered manually or whenever a commit or a pull request is made, in order to quickly see the results of any changes made to the Sikraken repository. By the time the Lambda is finished running, the user will receive a URL generated by the Lambda for the purpose of viewing the HTML report generated by one of the Sikraken scripts now created by a single Lambda.

GitHub Actions - [Pipeline Link]

This pipeline is run on GitHub Actions performs more tasks in comparison to the EC2 pipeline. This pipeline checks out the SikrakenDevOps repository, the Sikraken repository and the PTC Solver repository as it's needed for Sikraken to run [125] and the dockerfile and pipeline is currently located in the SikrakenDevOps repository. It also configures the AWS account credentials through OIDC with the same IAM role as the EC2 instance in order to update the task definition, which decides what containers are to be used, how much random access memory (RAM) and vCPUs are to be assigned for the task that the containers will perform .etc. Building and pushing the new Sikraken image to the Elastic Container Registry (ECR) for storing images, executing ECS tasks and lastly invoking the Lambda and retrieving its output also requires the configuration of AWS credentials.

Docker

Since ECS orchestrates containerised applications, it's necessary to containerise Sikraken and additionally the repository of benchmarks that Sikraken will process. Both containers share a volume, in which the benchmarks will copy its folders into the volume while Sikraken will wait for all benchmarks to show up before processing any of them to guarantee that there's no missing benchmarks in a test run.

(See Appendix 6 and 6.1 for the sequence diagram and script to generate a test run in the Sikraken container and upload the results)

Sikraken Container

The Sikraken container consists of all of its dependencies in order to install properly such as ECLiPSe Prolog as its environment runs Sikraken, AWS CLI so that the script within the container can upload its results to an S3.

The script stored inside the Sikraken container will use environment variables passed down from the container in order to configure the settings for Sikraken, the results visualisation options and also an integer ID for the vCPU that will run the containers as an ECS task. The script then searches for a .set file, which contains a list of .yml files for each benchmark to be run [126] before adding the name of each .yml file to an array. The array is looped through and only the indices whose remainder is equal to the ID of the vCPU after being modulo divided by the total number of vCPUs are processed by Sikraken to ensure that each Sikraken container is processing different benchmarks. Sikraken's results are then captured and uploaded to an S3 bucket.

Benchmarks Container

The benchmarks container simply consists of the C benchmarks from the benchmarks repository [79] and then copies the files to the volume shared with Sikraken so that they may be processed. This avoids using separate services that may add additional cost to the run such as using the Elastic File System [128] to act as storage for the benchmarks and then using the Sikraken container to process them, opting for copying the contents of the benchmarks container to the ephemeral storage that can be set up for each a task definition instead [113].

Elastic Container Registry (ECR)

The role of the ECR is to simply store Sikraken Docker images as well as the benchmarks container Docker image so that the ECS may pull the images to be used for a given number of ECS tasks that will be each run by a single vCPU. The registry is set to remove images that have not been pulled for 3 days to reduce the cost as the cost per GB stored is $0.10 each month [59].

While a new Sikraken image is stored for each pipeline run, the benchmarks container stays the same as it changes very little over the course of a year, preventing a several minute build time for its Docker image each time a new pipeline run takes place.

Elastic Container Service (ECS)

The ECS is the core of this implementation as this is where many vCPUs in a cluster will run a Sikraken container each. This starts by creating a task definition, which itself acts as a blueprint for how the containers will be run such as specifying the image ID of the Sikraken image and benchmarks image stored in the ECS so that they may be containerised, how much storage should be assigned (persistent or ephemeral), the number of vCPUs to run the containers and how much RAM is to be used alongside it as well as the operating system the containers should run on [64] as well as variables that can be passed down as environment variables in the containers [127].

A script is executed within the pipeline that specifies what cluster is to be used to run the containers and what task definition is to be used to specify how the containers should act. This script will run an individual ECS task, passing down an ID as an environment variable among others that configure the Sikraken container to handle the parallelism inside the container script. Upon running an individual task, a Sikraken and benchmarks container are run, starting the process of copying the benchmarks over to the shared volume and running Sikraken against the benchmarks.

(See Appendix 5 and 5.1 for the sequence diagram and script to execute ECS tasks)

Simple Storage Service (S3)

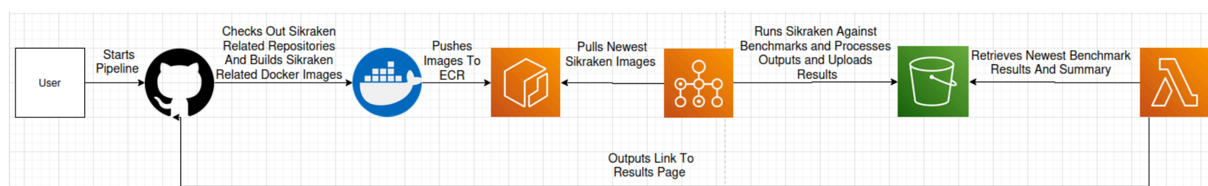
The simple storage service plays the same role as it does in the EC2 pipeline where it simply acts as cloud storage for folders generated after a test run and has additional metadata for easier processing within a Lambda function such as “last modified”.

AWS Lambda

Similar to the S3, the Lambda plays a very similar role in which it allows for code to be deployed to the cloud for a very cheap price and can be invoked through the pipeline and also have its output retrieved through it. While also used to retrieve the most recent benchmark results, it also generates a .html file to show the results of Sikraken’s performance for each benchmark where the url to this .html file is the Lambda’s output. This used to be done using a bash script, however since all the vCPUs in a cluster each run a part of the total number of benchmarks to process at the same time, waiting for the cluster to finish a test run and then visualising the results later seems to be the most practical solution.

(See Appendix 4 and 4.1 for the sequence diagram and script to generate a .html report for the Sikraken test run)

Architecture Diagram - AWS Batch



The above architecture diagram is a Batch implementation of the automation of running Sikraken against its benchmarks and would be cheaper than the ECS as it uses EC2 spot instances rather than Fargate vCPUs, leading to a much lower cost but potential interruptions due to the nature of spot instances.

Terraform

A Terraform configuration is used in order to allow the user to easily set up the AWS services required for the pipeline to function as they would simply need to install Terraform, clone the repository containing the Terraform configuration files and use three commands at most to initialise the configuration and create the AWS resources on their own account. The configuration is written in Hashicorp Configuration Language (HCL) as many of the examples documented for the modules and resources used in the configuration appear to be written in this language in comparison to JSON such as the documented examples in the Terraform Registry for AWS.

GitHub Actions - Pipeline Link

The AWS Batch pipeline also runs on GitHub Actions and follows nearly the exact same structure, with the main difference being that the script to initiate a run is now made for AWS Batch rather than ECS Fargate. This similarities that can be seen are that the same repositories are checked out, the same dockerfile is used and the process of building, tagging and pushing to ECR are the same, except the same image tag is used in the Batch pipeline rather than a hash provided by GitHub in

order to save costs as Docker images pushed with an existing tag in ECR tend to overwrite the existing image, reducing the amount of storage used in ECR.

Both pipelines also wait for jobs to be completed, and also send out a URL to reports generated by the test run to the user through a Lambda. The Batch pipeline has undergone further development in comparison to the ECS pipeline and therefore generates an extra URL through a Lambda to the user that summarises the results of previous test runs.

Various environment variables are used that can be set by the user in order to configure aspects of the test run such as, the number of jobs to run in parallel and the amount of memory to assign to each one. Environment variables are also used to allow the user to configure Sikraken for its test run such as the mode, the budget and the benchmarks category to run in.

Docker

Similar to ECS, AWS Batch also has a focus on containerised applications, requiring Sikraken's containerisation in this implementation. Creating a container for the benchmarks repository for Sikraken to process has now been avoided due to the use of spot instances as they can be interrupted and prolong the job to finish executing as the benchmarks would have to be transferred again to a shared volume if a similar architecture to the ECS implementation is used.

A new container consisting of Bash and Python scripts that are used to visualise the results of Sikraken test runs and also allow for the outputs to be used in the cloud is now used after all jobs using the Sikraken container are finished. Initially, the scripts to visualise results were run after a Sikraken test run was performed in the same script but since the script run inside the Sikraken container only runs a part of a category of benchmarks in comparison to all of them at once due to the fact that only one vCPU is used for each container, it's necessary to wait for all of them to be finished before collecting all of the results and processing them.

(See Appendix 13 and 14 for information on the Dockerfiles used to build the containers used)

AWS Batch

While both ECS and Batch make use of containerised applications that are capable of being run in parallel, ECS Fargate, the previous implementation was found to be expensive which prompted the use of EC2 spot instances as they are the cheapest option for cloud computing instances that AWS can offer. ECS with EC2 allows for the use of EC2 spot, it's required to manually manage the EC2 instances involved to allow Sikraken to process benchmarks as well as provision persistent storage to each one.

AWS Batch allows for the use of EC2 spot instances as well but much of the underlying infrastructure such as the EC2 instances used are provisioned and maintained by AWS but still provides the same benefits, simplifying the infrastructure overall and making it easier to work with. Batch still serves the same function as ECS in which it allows for a given number of vCPUs to be used to perform a given task or job which in this case is to run Sikraken against a given set of benchmarks.

Two jobs are run one after the other in this implementation, the first one being a Sikraken container that's run by multiple vCPUs to start a test run, and a reports container that starts running after all the array jobs that run the Sikraken container are finished. This allows for all the data regarding the test run to be outputted before any scripts are called to generate reports based on the test run results.

(See Appendix 3, 8, 9, 11, 12 for information on the scripts used to submit a job on Batch, and the scripts that are run inside the containers used in Batch)

Elastic Container Registry (ECR)

The ECR's role is the same as the one mentioned in the ECS implementation with the main difference being the removal of the benchmarks container. This is because of the fact that the AWS Batch implementation uses only spot instances, interruptions may cause the job to restart [134] and therefore making the process of transferring a large amount of benchmarks to a shared volume to also restart, prolonging the job.

Simple Storage Service (S3)

The simple storage service plays the same role as it does in the EC2 and ECS pipeline where it simply acts as cloud storage for folders generated after a test run with additional metadata for objects stored in an S3 bucket for easier processing within a Lambda function such as "last modified". Additionally, the S3 bucket is now used to hold benchmarks and only the necessary benchmarks are downloaded for each run within a Job, making this more efficient than uploading the entirety of the benchmarks in a shared Docker volume and only processing the few necessary.

AWS Lambda

The role played by the Lambda is very similar to the one in the EC2 implementation as it's purely used for retrieving the URL of the .HTML report of a test run to the user through the pipeline. In addition, the Lambda has been updated to also retrieve a .HTML summary of previous test runs to the user which is also performed within the pipeline.

(See Appendix 10 for information on the script run inside the Lambda function)

References

[1] Meudec, C. (2025). Sikraken: A Test Suites Generator for C Code (Github Repository)
Available At: https://github.com/echancrure/Sikraken?tab=readme-ov-file
[2] Test-Comp (2025) Benchmark Setup (Benchmark Results)
Available At:
https://test-comp.sosy-lab.org/2025/results/results-verified/META_Cover-Branches_sikraken.table.html#/
[3] GitHub (2025) Understanding GitHub Actions (Webpage)
Available At: https://docs.github.com/en/actions/get-started/understand-github-actions
[4] Biradar, M. & Moolya, S. (2022) Integrating with GitHub Actions – CI/CD pipeline to deploy a Web App to Amazon EC2
Available At:
https://aws.amazon.com/blogs/devops/integrating-with-github-actions-ci-cd-pipeline-to-deploy-a-web-app-to-amazon-ec2/
[5] Amazon Web Services (2025) Amazon EC2 (Webpage)
Available At: https://aws.amazon.com/ec2/
[6] Amazon Web Services (2025) What Is Amazon EC2? - Amazon Elastic Compute Cloud
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
[7] Amazon Web Services (2025) Amazon EC2 instance type specifications
Available At:
https://docs.aws.amazon.com/ec2/latest/instancetypes/ec2-instance-type-specifications.html
[8] Amazon Web Services (2025) CPU options for Amazon EC2 instances
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html
[9] Amazon Web Services (2025) Amazon EC2 instance type naming conventions
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/instance-type-names.html

[10] Amazon Web Services (2025) Specifications for Amazon EC2 general purpose instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/gp.html
[11] Amazon Web Services (2025) Amazon EC2 On-Demand Pricing
Available At: https://aws.amazon.com/ec2/pricing/on-demand/
[12] Amazon Web Services (2025) Specifications for Amazon EC2 compute optimized instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/co.html
[13] Amazon Web Services (2025) Specifications for Amazon EC2 memory optimized instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/mo.html
[14] Amazon Web Services (2025) Specifications for Amazon EC2 storage optimized instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/so.html
[15] Amazon Web Services (2025) Specifications for Amazon EC2 accelerated computing instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html
[16] Amazon Web Services (2025) Specifications for Amazon EC2 high-performance computing instances
Available At: https://docs.aws.amazon.com/ec2/latest/instancetypes/hpc.html
[17] Amazon Web Services (2025) Amazon Elastic Block Store
Available At: https://aws.amazon.com/ebs/
[18] Amazon Web Services (2025) Attach an Amazon EBS volume to an Amazon EC2 instance
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/ebs-attaching-volume.html
[19] Amazon Web Services (2025) Amazon EBS volume limits for Amazon EC2 instances
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/volume_limits.html
[20] Amazon Web Services (2025) Instance store temporary block storage for EC2 instances
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html
[21] Amazon Web Services (2025) Amazon EBS volume types
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/ebs-volume-types.html
[22] Amazon Web Service (2025) Amazon EBS I/O characteristics and monitoring
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/ebs-io-characteristics.html
[23] Amazon Web Service (2025) Amazon EBS pricing
Available At: https://aws.amazon.com/ebs/pricing/
[24] IEEE Explore (2019) TestCov: Robust Test-Suite Execution and Coverage Measurement
Available At: https://ieeexplore.ieee.org/document/8952265
[25] PyPi (2022) TestCov
Available At: https://pypi.org/project/TestCov/
[26] Amazon Web Services (2025) AWS Pricing Calculator
Available At: https://calculator.aws/#/addService
[27] CPU Benchmarks (2018) Intel Xeon Platinum 8175M Benchmark
Available At:
https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Platinum+8175M+%40+2.50GHz&id=3311
[28] CPU Benchmarks (2018) Intel Xeon Platinum 8124M Benchmark
Available At:
https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Platinum+8124M+%40+3.00GHz&id=3352
[29] CPU Benchmarks (2020) Intel Xeon Platinum 8259CL Benchmark
Available At:
https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Platinum+8259CL+%40+2.50GHz&id=3671
[30] CPU Benchmarks (2020) AMD EPYC 7R32 Benchmark
Available At: https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7R32&id=3894
[31] CPU Benchmarks (2023) AMD EPYC 9R14 Benchmark
Available At: https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+9R14&id=5635
[32] Test-Comp (2024) coverage-branches.ReachSafety-ECA – BenchExec results
Available At:
https://test-comp.sosy-lab.org/2025/results/results-verified/coverage-branches.ReachSafety-ECA.table.html#/table?pageSize=2500

[33] Rackspace (2025) About Rackspace Technology
Available At: https://www.rackspace.com/about
[34] Rackspace (2016) Current Offerings
Available At: https://docs.rackspace.com/docs/rackspace-elastic-engineering-and-optimizer
[35] Rackspace (2016) CloudHealth
Available At: https://docs.rackspace.com/docs/cloudhealth
[36] Meudec, C. (2023). PTC-Solver (Github Repository)
Available At: https://github.com/echancrure/PTC-Solver/tree/master
[37] SoSy-Lab (2025) TestCov
Available At: https://gitlab.com/sosy-lab/software/test-suite-validator
[38] Meudec, C. (2025) ECA Category Result
Available At: ECA_category_results.zip
[39] Sultan, R. (2025) Why is C Faster Than Python? OS-Level Perspective
Available At:
https://medium.com/@raihansltn/why-is-c-faster-than-python-os-level-perspective-04d6e485e3ae
[40] GeeksforGeeks (2023) cJSON JSON File Write/Read/Modify in C
Available At: https://www.geeksforgeeks.org/c/cjson-json-file-write-read-modify-in-c/
[41] Gamble, D. (2024) cJSON
Available At: https://github.com/DaveGamble/cJSON
[42] GeeksforGeeks (2019) Read JSON file using Python
Available At: https://www.geeksforgeeks.org/python/read-json-file-using-python/
[43] AWS (2025) Cloud Object Storage | Store & Retrieve Data Anywhere | Amazon Simple Storage Service
Available At: https://aws.amazon.com/s3/
[44] AWS (2025) What is Object Storage? - Object Storage - AWS
Available At: https://aws.amazon.com/what-is/object-storage/
[45] AWS (2025) Make an Amazon EBS volume available for use - Amazon EBS
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/ebs-using-volumes.html
[46] AWS (2025) Using Amazon S3 with Amazon EC2 - Amazon Elastic Compute Cloud
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonS3.html
[47] AWS (2013) AWS Command Line Interface
Available At: https://aws.amazon.com/cli/
[48] Kelley, K. (2024) What is AWS Lambda, and How Can You Write Functions and Code With it?
Available At: https://pg-p.ctme.caltech.edu/blog/cloud-computing/what-is-aws-lambda-functions-code
[49] AWS (2025) Lambda runtimes - AWS Lambda
Available At: https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html
[50] AWS (2025) Boto3 documentation — Boto3 Docs 1.40.53 documentation
Available At: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html#
[51] AWS (2025) get_object - Boto3 1.40.52 documentation
Available At:
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3/client/get_object.htm
l
[52] DataDog (2022) What Are Containerized Applications? | Datadog
Available At:
https://www.datadoghq.com/knowledge-center/containerized-applications/#what-are-containerized-ap
plications
[53] Docker Docs (2025) What is a container?
Available At: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/
[54] Docker Docs (2025) What is an image?
Available At: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/
[55] Docker Docs (2025) What is a registry
Available At: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-registry/

[56] Docker Docs (2025) Multi-container applications
Available At:
https://docs.docker.com/get-started/docker-concepts/running-containers/multi-container-applications/
[57] Dockers (2021) What is Docker Hub? | Docker
Available At: https://www.docker.com/products/docker-hub/
[58] Okonkwo, W. (2025) Deploying a Custom Web Application with Docker & Docker Compose on AWS EC2
Available At:
https://dev.to/teetoflame/deploying-a-custom-web-application-with-docker-docker-compose-on-aws-ec2-423n
[59] AWS (2024) Fully Managed Container Registry – Amazon Elastic Container Registry Pricing - Amazon Web Services
Available At: https://aws.amazon.com/ecr/pricing/
[60] Starr, S. (2023) Deploy a simple dockerized web app using AWS ECR & EC2
Available At:
https://medium.com/@sstarr1879/deploy-a-simple-dockerized-web-app-using-aws-ecr-ec2-e6a3569a6cf5
[61] JFrog (2022) Comparing Docker Hub and GitHub Container Registry
Available At:
https://jfrog.com/devops-tools/article/comparing-docker-hub-and-github-container-registry/
[62] Scale Computing (2025) What Is Terraform? Overview, Functions, and Benefits
Available At: https://www.scalecomputing.com/resources/what-is-terraform
[63] Nguyen, T. (2017) Gentle Introduction to How AWS ECS Works with Example Tutorial
Available At:
https://medium.com/boltops/gentle-introduction-to-how-aws-ecs-works-with-example-tutorial-cea3d27ce63d
[64] AWS (2025) Amazon ECS task definitions - Amazon Elastic Container Service
Available At: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html
[65] AWS (2025) Amazon ECS clusters - Amazon Elastic Container Service
Available At: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/clusters.html
[66] AWS (2025) Fully Managed Container Orchestration – Amazon ECS Pricing– Amazon Web Services
Available At: https://aws.amazon.com/ecs/pricing/
[67] Karishma (करिश्मा) Working with ECS.
Available At: https://medium.com/codex/working-with-ecs-cece5f447936
[68] Raina, A. (2024) How to Run AWS CLI in Docker - Collabnix
Available At: https://collabnix.com/how-to-run-aws-cli-in-docker/
[69] G, J M. (2023) Uploading Files to an S3 Bucket using AWS CLI
Available At:
https://medium.com/@josemanuel.gilperez/uploading-files-to-an-s3-bucket-using-aws-cli-4ac89a0b024b
[70] GeeksForGeeks (2025) Docker - COPY Instruction
Available At: https://www.geeksforgeeks.org/devops/docker-copy-instruction/
[71] Stack Overflow (2018) Can we include git commands in docker image?
Available At:
https://stackoverflow.com/questions/50870161/can-we-include-git-commands-in-docker-image
[72] AWS (2025) Connect to Your Linux Instance using an SSH client
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/connect-linux-inst-ssh.html
[73] Johanning, Josh. (2023) Using GitHub Actions Secrets to Store Certificates/Keys
Available At: https://josh-ops.com/posts/storing-certificates-as-github-secrets/
[74] GitHub Docs (2025) Secrets
Available At: docs.github.com/en/actions/concepts/security/secrets

[75] GitHub Docs (2025) Configuring OpenID Connect in Amazon Web Services
Available At:
https://docs.github.com/en/actions/how-tos/secure-your-work/security-harden-deployments/oidc-in-aws
[76] AWS (2025) What Is AWS Systems Manager?
Available At:
https://docs.aws.amazon.com/systems-manager/latest/userguide/what-is-systems-manager.html
[77] AWS (2025) send-command
Available At: https://docs.aws.amazon.com/cli/latest/reference/ssm/send-command.html
[78] AWS (2025) Amazon Machine Images in Amazon EC2
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
[79] SoSy Lab (2025) SoSy-Lab / Benchmarking / SV-Benchmarks · GitLab
Available At: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks
[80] AWS (2024) list-command-invocations — AWS CLI 2.32.3 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/ssm/list-command-invocations.html
[81] AWS (2015) list_objects_v2 - Boto3 1.35.5 documentation
Available At:
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3/client/list_objects_v2.html
[82] AWS (2024) Working with object metadata - Amazon Simple Storage Service
Available At: https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingMetadata.html
[83] AWS (2025) Sharing objects using presigned URLs - Amazon Simple Storage Service
Available At:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html
[84] AWS (2015) get_object - Boto3 1.40.52 documentation
Available At:
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3/client/get_object.html
[85] AWS (2025) invoke — AWS CLI 2.32.3 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/lambda/invoke.html#cli-binary-format
[86] AWS (2025) Start a session - AWS Systems Manager
Available At:
https://docs.aws.amazon.com/systems-manager/latest/userguide/session-manager-working-with-sessions-start.html
[87] AWS (2025) End a session - AWS Systems Manager
Available At:
https://docs.aws.amazon.com/systems-manager/latest/userguide/session-manager-working-with-sessions-end.html
[88] AWS (2025) mv — AWS CLI 2.32.3 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/s3/mv.html
[89] GeeksforGeeks (2019) Writing to file in Python
Available At: https://www.geeksforgeeks.org/python/writing-to-file-in-python/
[90] AWS (2025) start-instances — AWS CLI 2.32.6 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/ec2/start-instances.html
[91] AWS (2025) stop-instances — AWS CLI 1.27.26 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/ec2/stop-instances.html
[92] AWS (2022) instance-running — AWS CLI 2.32.6 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/ec2/wait/instance-running.html
[93] ECLiPSe (2025) ECLiPSe Home
Available At: https://eclipseclp.org/
[94] Docker Docs (2024) Building Best Practices
Available At: https://docs.docker.com/build/building/best-practices/

[95] Docker Docs (2024) Dockerfile reference
Available At: https://docs.docker.com/reference/dockerfile/
[96] Docker Docs (2024) Understanding the image layers
Available At:
https://docs.docker.com/get-started/docker-concepts/building-images/understanding-image-layers/
[97] Nayak, G R (2021) Why is DEBIAN_FRONTEND=noninteractive discouraged in Docker files?
Available At: https://bobcares.com/blog/debian_frontendnoninteractive-docker/
[99] AWS (2026) Install SSM Agent on Ubuntu Server 16.04 LTS 64-bit (Snap), 18.04, 20.04, 22.04 LTS, 23.10, 24.04 LTS, 24.0, and 25.04 - AWS Systems Manager
Available At:
https://docs.aws.amazon.com/systems-manager/latest/userguide/agent-install-ubuntu-64-snap.html
[99] Robotics Documentation (2025) Snap data and file storage
Available At:
https://canonical-robotics.readthedocs-hosted.com/en/latest/explanations/snaps/snap-data-and-file-storage/
[100] AWS (2018) sync — AWS CLI 2.9.6 Command Reference
Available At:
https://awscli.amazonaws.com/v2/documentation/api/2.9.6/reference/s3/sync.html#
[101] AWS (2026) What is Amazon Elastic Block Store? - Amazon EBS
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/what-is-ebs.html
[102] Padghan, V.(2018) AWS Fargate — A Beginner's Guide To AWS Elastic Container Service
Available At: https://medium.com/edureka/aws-fargate-85a0e256cb03
[103] AWS (2026) AWS Fargate Pricing
Available At: https://aws.amazon.com/fargate/pricing/
[104] ahmedjama.com (2025) ECS Managed Instances: A practical comparison with Fargate and EC2
Available At: https://ahmedjama.com/blog/2025/10/ecs/ecs-managed-instances-comparison/
[105] Docker Docs (2024) Volumes
Available At: https://docs.docker.com/engine/storage/volumes/
[106] GitHub Docs (2026) Deploying to Amazon Elastic Container Service - GitHub Docs
Available At:
https://docs.github.com/en/actions/how-tos/deploy/deploy-to-third-party-platforms/amazon-elastic-container-service
[107] AWS (2024) run-task — AWS CLI 2.33.23 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/ecs/run-task.html
[108] AWS (2026) Connect Amazon ECS applications to the internet - Amazon Elastic Container Service
Available At:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/networking-outbound.html #
[109] Peck, N. (2018) Task Networking in AWS Fargate | Amazon Web Services
Available At: https://aws.amazon.com/blogs/compute/task-networking-in-aws-fargate/
[110] AWS (2023) What Is Amazon VPC? - Amazon Virtual Private Cloud
Available At: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html
[111] AWS (2026) Set up to use Amazon ECS - Amazon Elastic Container Service
Available At:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/get-set-up-for-amazon-ecs.html
[112] AWS (2026) Default security groups for your VPCs - Amazon Virtual Private Cloud
Available At: https://docs.aws.amazon.com/vpc/latest/userguide/default-security-group.html
[113] AWS (2026) Storage options for Amazon ECS tasks - Amazon Elastic Container Service
Available At:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/using_data_volumes.html
[114] AWS (2019) Amazon Elastic File System (EFS) | Cloud File Storage | Pricing
Available At: https://aws.amazon.com/efs/pricing/

[115] AWS (2026) Use bind mounts with Amazon ECS - Amazon Elastic Container Service
Available At: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/bind-mounts.html
[116] AWS (2025) Amazon S3 Pricing
Available at: https://aws.amazon.com/s3/pricing/
[117] AWS (2025) Amazon EBS snapshots - Amazon EBS
Available At: https://docs.aws.amazon.com/ebs/latest/userguide/ebs-snapshots.html
[118] GitHub (2025) GitHub-hosted runners - GitHub Docs
Available At: https://docs.github.com/en/actions/concepts/runners/github-hosted-runners
[119] GitHub Docs (2025) Store information in variables - GitHub Docs
Available At:
https://docs.github.com/en/actions/how-tos/write-workflows/choose-what-workflows-do/use-variables
[120] AWS re:Post (2021) Calling AWS services with AWS SDK outside of AWS
Available At:
https://repost.aws/questions/QUXhdo3hWEQlSuhmVidopRVQ/calling-aws-services-with-aws-sdk-outside-of-aws
[121] AWS (2025) Credentials - Boto3 1.41.5 documentation
Available At: https://boto3.amazonaws.com/v1/documentation/api/latest/guide/credentials.html
[122] AWS (2024) AWS Lambda – Pricing
Available At: https://aws.amazon.com/lambda/pricing/
[123] Vercara (2025) Content-Type HTTP Header
Available At: https://vercara.digicert.com/resources/content-type-http-header
[124] AWS (2026) Common response headers - Amazon Simple Storage Service
Available At:
https://docs.aws.amazon.com/AmazonS3/latest/API/RESTCommonResponseHeaders.html
[125] Meudec, C. (2025) Sikraken Install, Development and User Guide
Available At:
https://docs.google.com/document/d/1uDLnlrFGWUNYyzsotZAZ_jFVrRSPEoixVMgw1UZZ0ug/edit?tab=t.0
[126] SoSy Lab (2026) c/ECA.set · main · SoSy-Lab / Benchmarking / SV-Benchmarks · GitLab
Available At:
https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/main/c/ECA.set?ref_type=heads
[127] AWS (2026) Pass an individual environment variable to an Amazon ECS container - Amazon Elastic Container Service
Available At: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/taskdef-envfiles.html
[128] AWS (2024) Amazon Elastic File System (EFS) | Cloud File Storage
Available At: https://aws.amazon.com/efs/
[129] EverythingDevOps (2024) What Is Amazon Resource Name (ARN)?
Available At: https://www.everythingdevops.dev/blog/what-is-amazon-resource-name-arn
[130] AWS (2024) Amazon ECS task lifecycle - Amazon Elastic Container Service
Available At:
https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-lifecycle-explanation.html [131] AWS (2019) AWS Batch — Easy and Efficient Batch Computing Capabilities - AWS
Available At: https://aws.amazon.com/batch/
[132] AWS (2026) AWS Batch Pricing
Available At: https://aws.amazon.com/batch/pricing/
[133] AWS (2025) Tutorial: Create a managed compute environment using Amazon EC2 resources - AWS Batch
Available At:
https://docs.aws.amazon.com/batch/latest/userguide/create-compute-environment-managed-ec2.html
[134] AWS (2026) Spot Instances - Amazon Elastic Compute Cloud
Available At: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html
[135] AWS (2019) Amazon EC2 Spot Instances Pricing

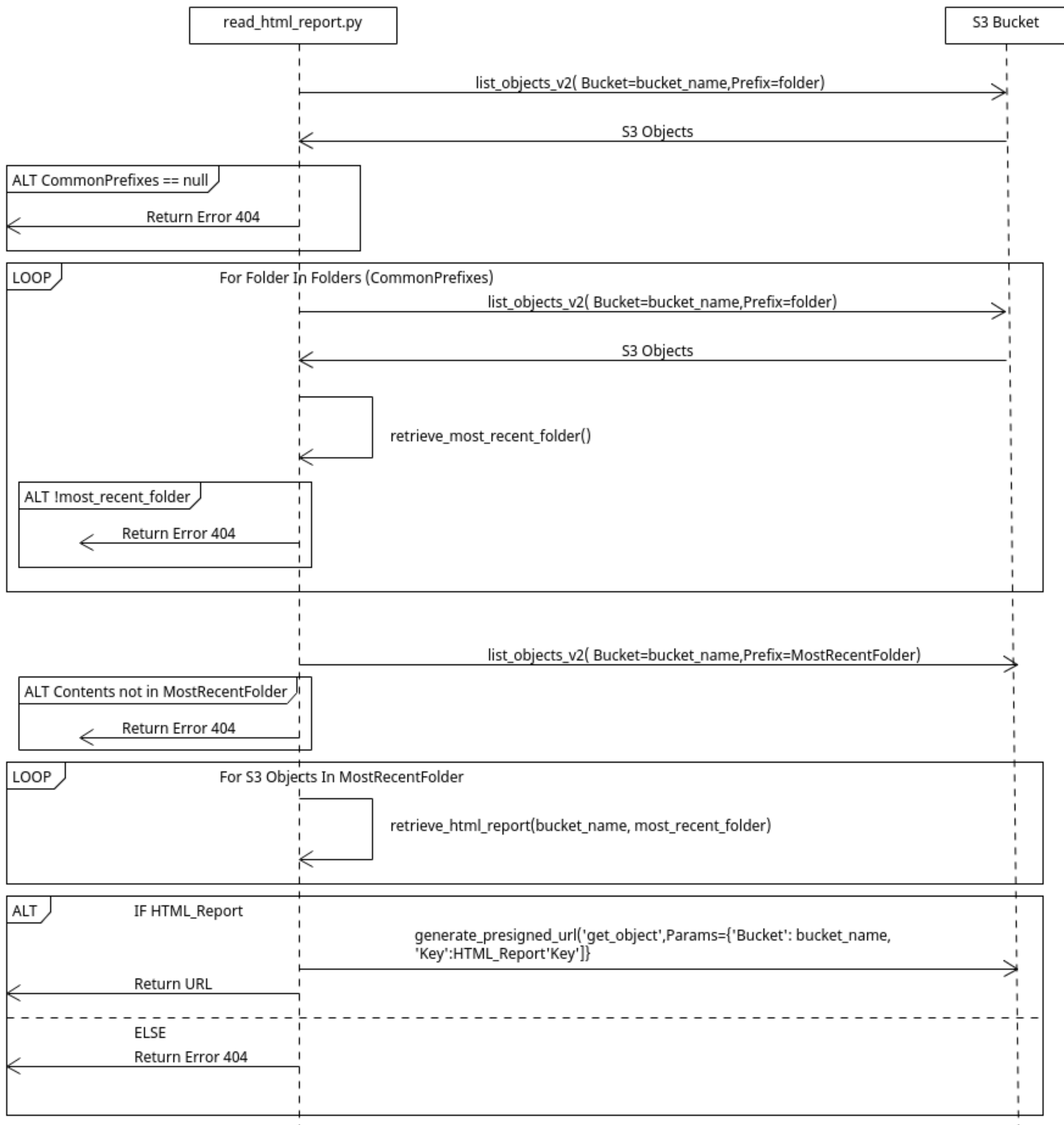
Available At: https://aws.amazon.com/ec2/spot/pricing/
[136] AWS (2026) Job definitions - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/job_definitions.html
[137] AWS (2026) Jobs - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/jobs.html
[138] AWS (2026) AWS Batch job environment variables - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/job_env_vars.html
[139] AWS (2026) Array Jobs - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/array_jobs.html
[140] AWS (2026) Job dependencies - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/job_dependencies.html
[141] AWS (2026) Job queues - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/job_queues.html
[142] AWS (2026) Compute environments for AWS Batch - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/compute_environments.html
[143] AWS (2026) AWS Batch FAQs
Available At: https://aws.amazon.com/batch/faqs/
[144] Awad, J W. (2025) AWS ECS Deep Dive
Available At: https://joudwawad.medium.com/aws-ecs-deep-dive-c8f773af0bf6
[145] AWS (2023) Amazon EC2 container instances for Amazon ECS - Amazon Elastic Container Service
Available At: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-capacity.html
[146] AWS (2026) Job states - AWS Batch
Available At: https://docs.aws.amazon.com/batch/latest/userguide/job_states.html
[147] HashiCorp (2024) Create infrastructure | Terraform | HashiCorp Developer
Available At: https://developer.hashicorp.com/terraform/tutorials/aws-get-started/aws-create
[148] AWS (2026) s3 — AWS CLI 1.31.10 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/s3/
[149] HashiCorp (2025) Resources Overview - Configuration Language | Terraform | HashiCorp Developer
Available At: https://developer.hashicorp.com/terraform/language/resources
[150] HashiCorp (2025) Providers - Configuration Language | Terraform | HashiCorp Developer
Available At:
https://developer.hashicorp.com/terraform/language/providers
[151] HashiCorp (2025) Modules Overview - Configuration Language | Terraform | HashiCorp Developer
Available At: https://developer.hashicorp.com/terraform/language/modules
[152] AWS (2025) What Is IAM? - AWS Identity and Access Management
Available At: https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html
[153] AWS (2019) Identities (Users, Groups, and Roles) - AWS Identity and Access Management
Available At: https://docs.aws.amazon.com/IAM/latest/UserGuide/id.html
[154] AWS (2026) IAM roles - AWS Identity and Access Management
Available At: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html#iam-term-service-role
[155] AWS (2026) submit-job — AWS CLI 2.34.5 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/batch/submit-job.html
[156] Terraform Registry (2026) AWS Batch Terraform module
Available At: https://registry.terraform.io/modules/terraform-aws-modules/batch/aws/latest
[157] Terraform Registry (2026) Amazon ECR Terraform module
Available At: https://registry.terraform.io/modules/terraform-aws-modules/ecr/aws/latest
[158] Terraform Registry (2026) Amazon Lambda Terraform module
Available At: https://registry.terraform.io/modules/terraform-aws-modules/lambda/aws/latest
[159] Terraform Registry (2026) Amazon S3 bucket Terraform module

Available At: https://registry.terraform.io/modules/terraform-aws-modules/s3-bucket/aws/latest
[160] Terraform Registry (2026) Resource: aws_iam_role
Available At: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/iam_role
[161] Terraform Registry (2026) Resource: aws_iam_role_policy
Available At:
https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/iam_role_policy
[162] Terraform Registry (2026) AWS IAM Terraform module
Available At: https://registry.terraform.io/modules/terraform-aws-modules/iam/aws/latest
[163] AWS (2019) describe-jobs — AWS CLI 2.34.21 Command Reference
Available At: https://docs.aws.amazon.com/cli/latest/reference/batch/describe-jobs.html
[164] AWS (2026) Amazon S3 objects overview - Amazon Simple Storage Service
Available At: https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingObjects.html
[165] Ubuntu (2026) Install a root CA certificate in the trust store
Available At:
https://ubuntu.com/server/docs/how-to/security/install-a-root-ca-certificate-in-the-trust-store/
[166] askUbuntu (2016) What is the use/purpose of the ca-certificates package?
Available At:
https://askubuntu.com/questions/857476/what-is-the-use-purpose-of-the-ca-certificates-package
[167] AWS (2026) Working with Lambda layers - AWS Lambda
Available At: https://docs.aws.amazon.com/lambda/latest/dg/chapter-layers.html
[168] AWS (2024) Amazon CloudWatch - Application and Infrastructure Monitoring
Available At: https://aws.amazon.com/cloudwatch/

Appendix

Appendix 1: Lambda Script Sequence Diagram

The below sequence diagram details the script within the AWS Lambda to find the .html report within the folder uploaded to the S3 bucket.



Appendix 1.1: Lambda Script Code

Below is the Python script used inside the Lambda to find the .html report within the folder uploaded to the S3 bucket by finding through an object's key, which is essentially the object's name [164].

```
import json
import boto3
```

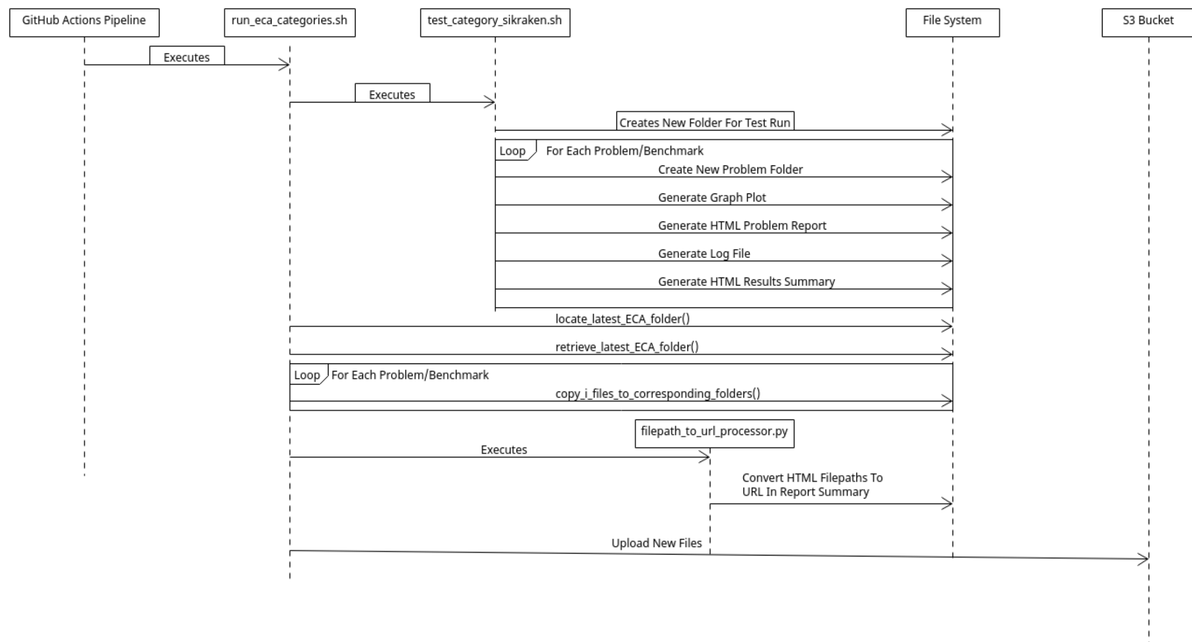
```
# Initialize S3 client
s3 = boto3.client('s3')

def lambda_handler(event, context):
   bucket_name = 'testcov-results-bucket'
   object_key = 'category_test_run_results.html'  # File to search for

   # List objects with the delimiter '/' to only show directories
   response = s3.list_objects_v2(
       Bucket=bucket_name,
       Delimiter='/'
   )

   # CommonPrefixes is a list of dictionaries of prefixes which are the
names given to the S3 objects/files in the S3  directory, if it's
empty, it means there's no files
   if 'CommonPrefixes' not in response:
       return {
           'statusCode': 404,
           'body': json.dumps('No folders found in the bucket.')
       }

   # The list of directory names is the value to the key "prefix"
stored in the list of dictonaries in CommonPrefixes which will be used
to find the most recently created folder
   folders = [prefix['Prefix'] for prefix in
response['CommonPrefixes']]

   print("Folders: ",folders)

   most_recent_folder = None
   most_recent_time = None

   # Iterate through each folder to find the most recent file which
should be the .html file
   for folder in folders:
       # List objects within the folder to find the most recent file
       folder_response = s3.list_objects_v2(
           Bucket=bucket_name,
           Prefix=folder
       )

       if 'Contents' in folder_response:
```

```
            # Get the most recent object in this folder, max gets the
folder with the highest value/Last Modified Date and passes the value
of contents to the lambda
            recent_object = max(folder_response['Contents'], key=lambda
x: x['LastModified']) #The anonymous lambda function finds the most
recent folder by comparing the last modified files passed from Contents
            folder_last_modified = recent_object['LastModified']
            print("FOLDER RESPONSE CONTENTS:",
folder_response['Contents']) #Print line used when debugging
            # Compare with the most recent folder we've seen
            if most_recent_time is None or folder_last_modified >
most_recent_time:
                most_recent_folder = folder
                most_recent_time = folder_last_modified

    if not most_recent_folder: #Returning an error if there's nothing in
the folders
        return {
            'statusCode': 404,
            'body': json.dumps('No folder found with objects in it.')
        }

    #Listing the contents of the most recent folder found to find the
HTML file
    folder_contents_response = s3.list_objects_v2(
        Bucket=bucket_name,
        Prefix=most_recent_folder
    )

    #Check if 'Contents' is present in the folder's response
    if 'Contents' not in folder_contents_response:
        return {
            'statusCode': 404,
            'body': json.dumps(f'No contents found in folder
{most_recent_folder}.')
        }

    # Checking for object with the name of report generated
    html_report = [obj for obj in folder_contents_response['Contents']
if obj['Key'].endswith(object_key)]

    if html_report:
```

```
        file_obj = html_report[0] #Getting first S3 object found that 
has the correct key name as a list is returned
        # Generate a pre-signed URL for the specific HTML file 
(optional)
        url = s3.generate_presigned_url(
            'get_object',
            Params={'Bucket': bucket_name, 'Key': file_obj['Key']}, 
#Getting key/name of that object returned
            ExpiresIn=3600  # URL valid for 1 hour, presigned URLs have 
expiration, will find a way around this in the future
        )

        return {
            'statusCode': 200,
            'body': json.dumps({
                'message': f"Found the HTML file: {file_obj['Key']}",
                'url': url  #.html file url from S3 bucket that can be 
viewed by anyone with the link
            })
        }
    else:
        return {
            'statusCode': 404, # Error message in the case that
            'body': json.dumps(f'{object_key} not found in folder 
{most_recent_folder}.')
        }
```

Appendix 2: EC2 Instance Script Sequence Diagram

The sequence diagram below details the current logic for the bash script used within the EC2 instance.



Appendix 2.1: EC2 Instance Script Code

The script below is the bash script called using the send-command actions using SSM from the pipeline. The script takes the command line arguments and runs an existing script on the SikrakenDevSpace repository to run Sikraken against the ECA benchmark. The script then finds the folders generated by the existing script, finds the latest test run using Regular Expression and the find command and then assigning its basename to a variable for the Python script to process .html report found within it.

The latest ECA folder is then placed into a different directory in the EC2 instance as the folders in the snap environment are temporary. The .i files created by Sikraken are then copied into their corresponding folders before the Python script for processing the .html report's filepaths is executed. The folder will then be uploaded to the S3 bucket, starting with all the files excluding the .i and .log files and then the .i and .log files. This is because the content-type is specified for the sync operation [100], meaning that if the content-type is set to text/plain, all the files will be set to that content.

The content type in a HTTP refers to the type of media in a HTTP header that decides how the content will be handled such as displaying text based content for a text/plain header. [123]. Setting the content-type for an S3 object changes content type of the HTTP header. [124]

```bash
#!/bin/bash
set -euo pipefail

category="$1"
cores="$2"
budget="$3"
mode="$4"
testcov_switch="$5"
stack_size="$6"
```

```
SIKRAKEN_DEVSPACE="/home/ubuntu/Sikraken/SikrakenDevSpace"
SIKRAKEN_OUTPUT_PATH="/home/ubuntu/Sikraken/sikraken_output"
CATEGORY_END_LOCATION="/home/ubuntu/SikrakenDevSpace/categories/ECA"
#Location where all ECA test runs will be stored to prevent them
PYTHON_SCRIPTS="/home/ubuntu/Sikraken/SikrakenDevOps/SikrakenPythonSc
ripts"

echo "Running Sikraken test..." #Running test_category_sikraken
"$SIKRAKEN_DEVSPACE/bin/test_category_sikraken.sh" \
   "/home/ubuntu/Sikraken/sv-benchmarks/c" \
   "$category" "$cores" "$budget" "$mode" "$testcov_switch"
"$stack_size"

#Function for finding the latest ECA folder
locate_latest_ECA_folder(){
   echo "Locating generated ECA run folder in SSM sandbox..."

   SSM_BASE="/var/snap/amazon-ssm-agent" #Starting path for ECA
folder results as the ECA folders will be stored within the snap
environment
   SSM_CATEGORIES="$SSM_BASE/*/SikrakenDevSpace/categories/ECA"
#Using globbing to collect all folders matching the format (The *
matches with anything and is used as IDs are generated within the
file path by the snap environment when a new one is created)

   # Find the most recent timestamped folder using Regex and then
sort and tail to get the most recent folder. The ".*/" will match
with the file path found before.
   # mindepth 1 and maxdepth 1 ensures that only the children (The
timestamped folders are processed) and -type d only matches
directories
   FOUND_ECA_DIR=$(find $SSM_CATEGORIES -mindepth 1 -maxdepth 1 -type
d \
       -regextype posix-extended \
       -regex ".*/[0-9]{4}_[0-9]{2}_[0-9]{2}_[0-9]{2}_[0-9]{2}" \
       | sort | tail -n1)

   if [[ -z "$FOUND_ECA_DIR" ]]; then
       echo "ERROR: Could not find the timestamped ECA folder"
       exit 1
   fi
```

```
    RUN_FOLDER=$(basename "$FOUND_ECA_DIR") #Folder for Python script
to process .html report
    TARGET_BASE="$FOUND_ECA_DIR"

    echo "Found timestamped ECA folder: $TARGET_BASE"
    echo "Run folder name: $RUN_FOLDER"
}

locate_latest_ECA_folder

retrieve_latest_ECA_folder(){
    mkdir -p "$CATEGORY_END_LOCATION" #Creating new ECA folder if it
doesn't exist and then moving the ECA folder created to this folder

    if [[ "$TARGET_BASE" != "$CATEGORY_END_LOCATION/$RUN_FOLDER" ]];
then
        echo "Moving ECA folder to final location..."
        sudo mv "$TARGET_BASE" "$CATEGORY_END_LOCATION/"
    fi

    sudo chown -R ubuntu:ubuntu "$CATEGORY_END_LOCATION/$RUN_FOLDER"
#Using chown for ubuntu as it belongs to root otherwise

    TARGET_BASE="$CATEGORY_END_LOCATION/$RUN_FOLDER"
    }
retrieve_latest_ECA_folder

copy_i_files_to_corresponding_folders(){
    echo "Copying .i files..."
    for d in "$TARGET_BASE"/*/; do
        name=$(basename "$d")
        src_file="$SIKRAKEN_OUTPUT_PATH/$name/$name.i"
        if [[ -f "$src_file" ]]; then
        cp "$src_file" "$d"
        echo "Copied $src_file → $d"
        else
        echo "Missing: $src_file"
        fi
    done
}
copy_i_files_to_corresponding_folders
```

```
echo "Processing $TARGET_BASE for S3..."
python3 "$PYTHON_SCRIPTS/filepath_to_url_processor.py" "$TARGET_BASE"
--run_folder "$RUN_FOLDER"

upload_folder_to_S3(){
# Uploading ECA files excluding .i and .log to S3.
echo "Syncing non-text files to S3..."
aws s3 sync "$TARGET_BASE" "s3://temp-bucket-sikraken/$RUN_FOLDER" \
 --exclude "*.i" \
 --exclude "*.log"

# Uploading ECA files including .i and .log to S3. The content-type
is applied to all files being uploaded requiring seperate uploads.
Changing the content type allows the files to be viewable in browser.
echo "Syncing .i files as text/plain..."
aws s3 sync "$TARGET_BASE" "s3://temp-bucket-sikraken/$RUN_FOLDER" \
 --exclude "*" \
 --include "*.i" \
 --include "*.log" \
 --content-type text/plain
}
upload_folder_to_S3

echo "Pipeline completed successfully!"
```

Appendix 3: Filepath Processor Python Script

The script below is the last script used in the pipeline and requires the most recent ECA folder generated for easier processing when uploading to the S3 bucket, as well as the full path to the most recent ECA folder created for further processing. It starts by stripping the “/” at the end of the directory name of the run folder (the name of the most recently generated folder after running Sikraken against the ECA categories), and then setting up the regex pattern to search for within the .html report. The replacement takes place at the return statement when re.sub will search for the pattern and then process the line/filepath with the S3 URL instead. The new file data is then appended to a list, where the first index is the file name and the second index is the new .html data, both of which will be outputted at the end but the actual .html file is opened, and then the new data is written to it.

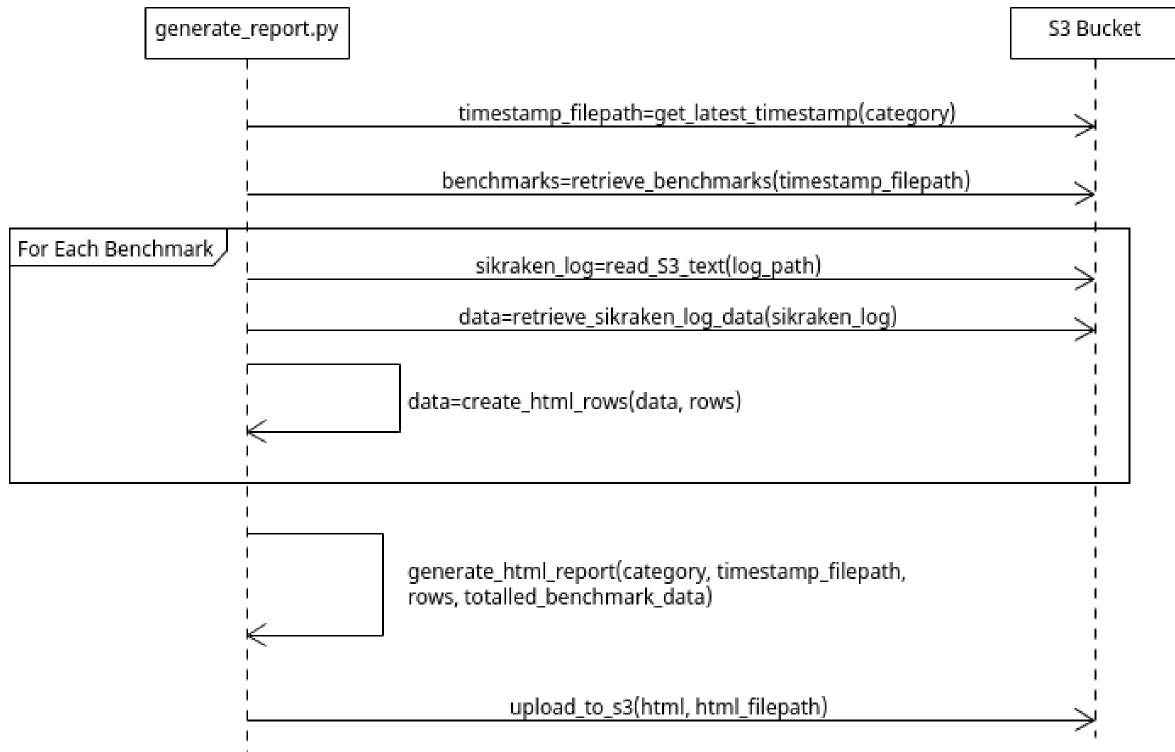
```
import os
import re
import json
import argparse
from pathlib import Path
```

```
s3_url = 'https://testcov-results-bucket.s3.eu-west-1.amazonaws.com'

def replace_local_paths_with_s3(html_content, run_folder):
   run_folder = run_folder.rstrip("/")  #Removing forward slash at end
of folder name

   pattern = r'href="(?!https?://)([^"]+)"' #Regex for detecting href
and doesn't already contain https

   def repl(match): #Matches by the pattern are used as a parameter in
this function once they're found
       path = match.group(1) #Using first match

       if path.startswith("file://"):
           path = path.replace("file://", "", 1) #ELiminating the file
path

       path = os.path.normpath(path) #Eliminate double slashes in path
if any
       p = Path(path) #Turns into Path object that has methods to
perform operations on it
       filename = p.name #Getting file name
       problem_folder = p.parent.name #Getting parent folder name to
match object format in S3

       relative_path = f"{run_folder}/{problem_folder}/{filename}"
#Combining the name e.g
2025_11_17_19_00/Problem03_label00/Problem03_label00.i
       return f'href="{s3_url}/{relative_path}"' #Adding to S3 url

   return re.sub(pattern, repl, html_content) #Scans for the pattern in
html_content and then calls repl to process the filepath with each
pattern found

def process_html_file(input_dir, run_folder):
   input_dir_path = Path(input_dir)
   html_file_name = 'category_test_run_results.html'
   html_full_path = os.path.join(input_dir_path, html_file_name)

   html_files = list(input_dir_path.glob(html_file_name)) #Get all
files with the name category_test_run_results.html in the input
directory to count the number of files processed
   report_data = []
```

```
    try:
        with open(html_full_path, 'r') as file: #Read file
            html_content = file.read()
        converted_html = replace_local_paths_with_s3(html_content,
run_folder) #Read html contents and replace file paths with S3 in order
to link to objects in bucket
        report_data.append({ #Appending updated data to the html
            "file": str(html_full_path),
            "content": converted_html
        })

    except Exception as e:
        return {"body": f"Error reading file {html_full_path}:
{str(e)}"}

    with open(html_full_path, 'w') as f:
        f.write(converted_html)

    return {"body": f"Processed {len(html_files)} HTML files.", "data":
report_data}

def main():
    parser = argparse.ArgumentParser(description="Generate a report from
test logs.") #Getting Arguments
    parser.add_argument('input_dir', type=str, help="Path to the input
directory") #Requiring input directory (ECA Category Folder)
    parser.add_argument('--run_folder', type=str, required=True,
help="Run folder name to use in S3 URLs") #Sets the folder name that
will be used in the S3 bucket

    args = parser.parse_args()
    result = process_html_file(args.input_dir, args.run_folder)

    print(result['body'])
    if 'data' in result:
        print(json.dumps(result['data'], indent=4))

if __name__ == "__main__":
    main()
```

Appendix 4: Lambda Script Sequence Diagram

The below sequence diagram details the logic for the Lambda script for the ECS implementation of the pipeline in order to read the Sikraken logs and generate a .html report of the Sikraken test run results from the data read from the logs.



Appendix 4.1: Lambda Script Code

The Python code below takes advantage of the boto3 SDK in order to interact with a given S3 bucket. It makes use of the list_objects_v2 function provided by the SDK [81] in order to retrieve the objects in the bucket using a delimiter and a prefix and returns a list known as “CommonPrefixes”, which contain the keys of each object that represents a file path in the bucket [84]. The prefix is the category of the test run in the pipeline and by using it with a delimiter, all the test runs of that specific category are shown as the prefix limits the search to objects that begin with the prefix e.g /ECA has a “/” delimiter and a prefix of ECA.

By retrieving all the benchmarks the most latest can be found by placing the keys into a list, sorting them by name, reversing the sort, and then picking the first option. Once the latest object is found, the sikraken.log for each benchmark run is read and used to build a .html report which will have its url return to the pipeline by the Lambda.

```
import boto3

import re

BUCKET = "ecs-benchmarks-output"

s3 = boto3.client("s3")
```

```
def retrieve_benchmarks(prefix):
   resp = s3.list_objects_v2(
       Bucket=BUCKET,
       Prefix=prefix,
       Delimiter="/"
   )
   return [p["Prefix"] for p in resp.get("CommonPrefixes", [])]
def get_latest_timestamp(category):
   prefixes = retrieve_benchmarks(f"{category}/")
   timestamp_filepaths = [p.rstrip("/").split("/")[-1] for p in
prefixes]
   timestamp_filepaths.sort(reverse=True)
   return timestamp_filepaths[0]
def read_s3_text(key):
   obj = s3.get_object(Bucket=BUCKET, Key=key)
   return obj["Body"].read().decode("utf-8", errors="ignore")
def retrieve_sikraken_log_data(text):
   lines = text.splitlines()
   def last_match(pattern):
       for line in reversed(lines):
           m = re.search(pattern, line)
           if m:
               return m.group(1)
       return None

   coverage = last_match(r"Coverage:\s+([\d.]+)%")
   if coverage is None:
```

```
        coverage = last_match(r"Inter-cov:\s*([\d.]+)%")


    tests = last_match(r"Generated:\s+(\d+)")

    cpu = last_match(r"times:\s*\[([\d.]+)")

    stack = last_match(r"global_stack_peak:\s+(\d+)")

    return {

        "coverage": float(coverage) if coverage else -1,

        "tests": int(tests) if tests else 0,

        "cpu": float(cpu) if cpu else 0.0,

        "stack_mb": round(int(stack) / 1_000_000, 2) if stack else 0.0,

    }

def generate_html_report(category, timestamp_filepath, rows,
totalled_benchmark_data):

    html_data = f"""<!DOCTYPE html>

<html>

<head>

<title>{category} Sikraken Results</title>

<style>

table {{

 border-collapse: collapse;

 width: 100%;

}}

th, td {{

 border: 1px solid black;

 padding: 6px;

}}

th {{
```

```
 background-color: #f2f2f2;

}}

</style>

</head>

<body>

<h1>Category: {category}</h1>

<h2>Timestamp: {timestamp_filepath}</h2>

<h2>Total Tests: {totalled_benchmark_data["tests"]}</h2>

<h2>Total CPU Time: {totalled_benchmark_data["cpu"]}</h2>

<table>

<tr>

 <th>Benchmark</th>

 <th>Sikraken Log</th>

 <th>Tests</th>

 <th>Coverage</th>

 <th>Stack Peak (MB)</th>

 <th>User CPU</th>

</tr>

{''.join(rows)}

</table>

</body>

</html>

"""

   html_filepath =
f"{category}/{timestamp_filepath}/category_test_run_results.html"

   return html_data, html_filepath

def upload_to_s3(html_data, html_filepath):
```

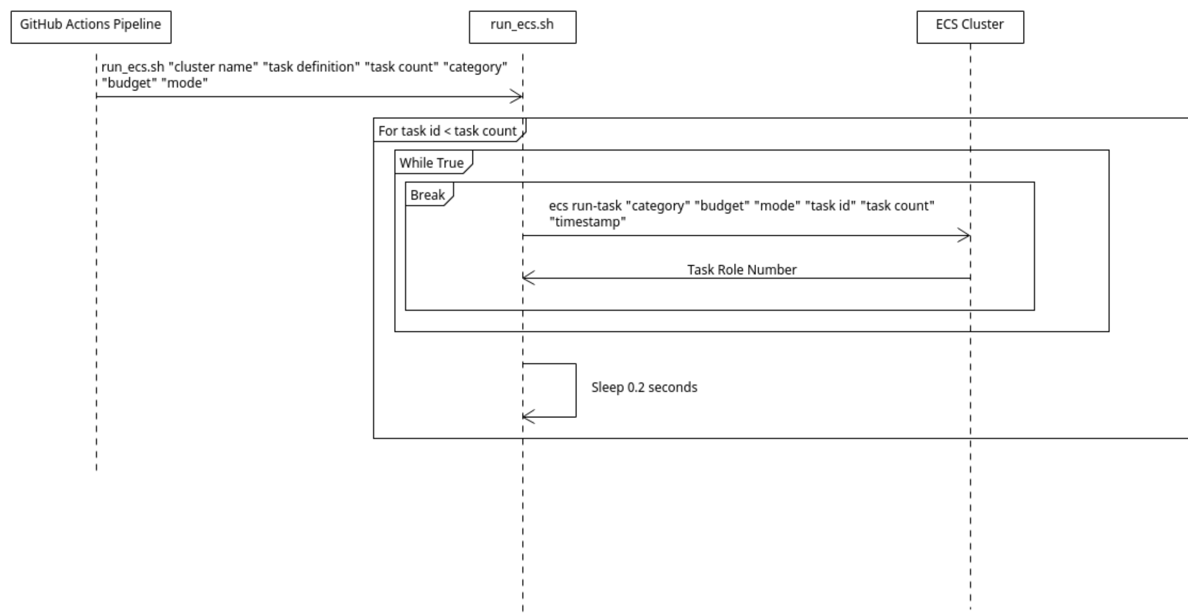
```
    s3.put_object(
        Bucket=BUCKET,
        Key=html_filepath,
        Body=html_data.encode("utf-8"),
        ContentType="text/html"
    )
    html_url = f"https://{BUCKET}.s3.amazonaws.com/{html_filepath}"
    return html_url
def retrieve_benchmark_name(benchmark):
    name = benchmark.rstrip("/").split("/")[-1]
    return name
def build_log_path(benchmark):
    log_path = f"{benchmark}sikraken.log"
    return log_path
def build_log_url(log_path):
    log_url = f"https://{BUCKET}.s3.amazonaws.com/{log_path}"
    return log_url
def create_html_rows(totalled_benchmark_data, data, rows,
log_url,name):
    totalled_benchmark_data["tests"] += data["tests"]
    totalled_benchmark_data["cpu"] += data["cpu"]

    rows.append(f"""
    <tr>
    <td>{name}</td>
    <td><a href="{log_url}" target="_blank">Sikraken Log</a></td>
    <td>{data["tests"]}</td>
```

```
    <td>{data["coverage"]}%</td>

    <td>{data["stack_mb"]}</td>

    <td>{data["cpu"]}</td>

    </tr>

    """)

    return rows, totalled_benchmark_data

def generate_rows(benchmarks):

    totalled_benchmark_data = {"coverage": 0.0, "tests": 0, "cpu": 0.0}

    rows = []

    for benchmark in benchmarks:

        name = retrieve_benchmark_name(benchmark)

        log_path = build_log_path(benchmark)

        log_url = build_log_url(log_path)

        log_text = read_s3_text(log_path)


        data = retrieve_sikraken_log_data(log_text)

        rows, totalled_benchmark_data =
create_html_rows(totalled_benchmark_data, data, rows, log_url, name)

    return rows, totalled_benchmark_data

def lambda_handler(event, context):

    category = event["Category"]

    timestamp_filepath = get_latest_timestamp(category)

    base_prefix = f"{category}/{timestamp_filepath}/"

    benchmarks = retrieve_benchmarks(base_prefix)

    rows, totalled_benchmark_data = generate_rows(benchmarks)

    html_data, html_filepath = generate_html_report(category,
timestamp_filepath, rows, totalled_benchmark_data)

    html_url = upload_to_s3(html_data, html_filepath)
```

```
return {
    "statusCode": 200,
    "html_url": html_url,
    "timestamp": timestamp_filepath,
    "benchmarks": len(rows)
}
```

Appendix 5: Run ECS Task Script Sequence Diagram

The below sequence diagram details the logic for a bash script executed by a GitHub runner in order to launch ECS tasks within a cluster with given environment variables to configure the containerised Sikraken repository.



Appendix 5.1: Run ECS Task Script

The script below takes advantage of AWS CLI's ecs run task command in order to launch an ECS task. The variables declared such as the cluster name and security groups and subnets are for configuring the task. Variables such as category, budget, mode are used to configure the Sikraken settings and are passed down as environment variables in the Sikraken container.

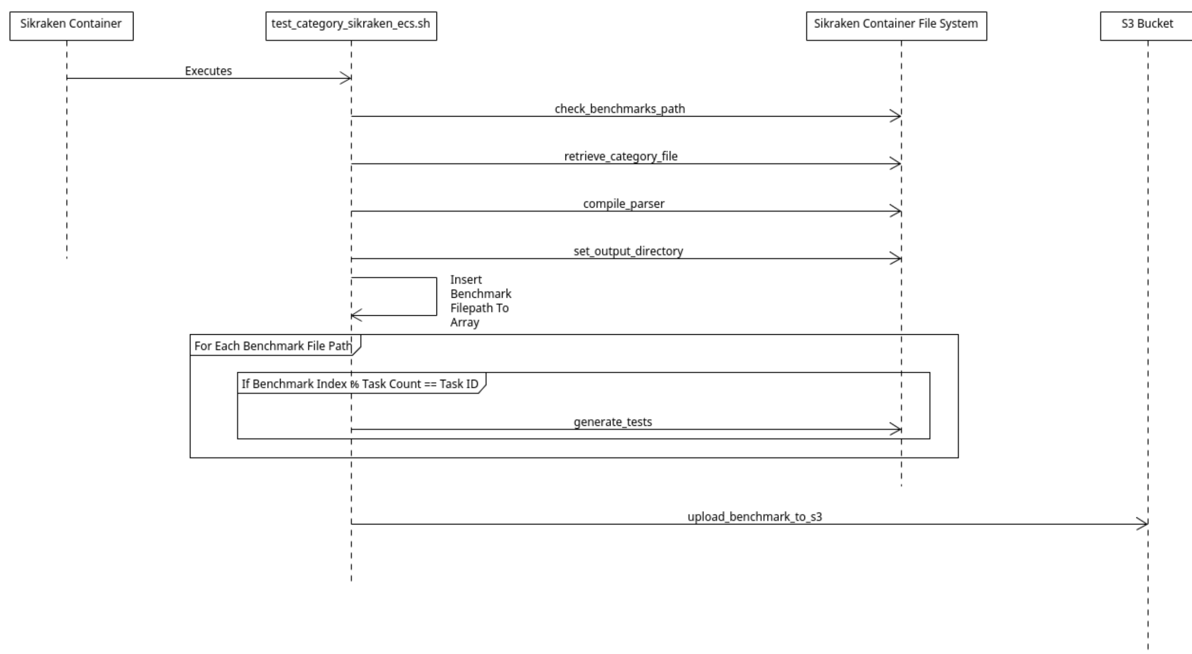
The tasks are launched one by one in a for loop with a 0.2 second delay between each launch. The reason for this is that a limit is set by AWS on how many tasks can be launched rapidly so a delay is used to bypass this. While the ecs run task command allows for a specific number of tasks to be launched at once, it wouldn't allow for a task ID to be passed down as an environment variable for further concurrent work in the script run inside the Sikraken container. The script also gets the output of each run task command, an Amazon Resource Name (ARN), which simply acts as an ID for AWS resources such as an ECS task [129], and if the length of the ARN outputted is non-zero, it means it's successful, otherwise it's not and the while loop will cause the launch to be retried.

```
#!/usr/bin/env bash
set -euo pipefail

CLUSTER="${1:-${CLUSTER:-sikraken-cluster}}"
TASK_DEF="${2:-${TASK_DEF:-sikraken-benchmarks-task-def}}"
TASK_COUNT="${3:-${TASK_COUNT:-5}}"
CATEGORY="${4:-${CATEGORY:-chris}}"
BUDGET="${5:-${BUDGET:-10}}"
MODE="${6:-${MODE:-release}}"
TIMESTAMP=$(date -u +"%Y_%m_%d_%H_%M")

SUBNET_ARRAY=(subnet-00575f764f10645c4 subnet-0d48c3c69206076d1
subnet-0a693be6424dd272a)
SG="sg-0b94b75a72c6f0356"

TASK_ARNS=()

launch_task() {
   local TASK_INDEX=$1
   local SUBNET=${SUBNET_ARRAY[$((TASK_INDEX % ${#SUBNET_ARRAY[@]}))]}

   OUT=$(aws ecs run-task \
       --cluster "$CLUSTER" \
       --task-definition "$TASK_DEF" \
       --launch-type FARGATE \
       --count 1 \
       --overrides "{
         \"containerOverrides\": [
           {
             \"name\": \"sikraken-container\",
             \"environment\": [
               {\"name\": \"CATEGORY\", \"value\": \"$CATEGORY\"},
               {\"name\": \"BUDGET\", \"value\": \"$BUDGET\"},
               {\"name\": \"MODE\", \"value\": \"$MODE\"},
               {\"name\": \"TASK_INDEX\", \"value\": \"$TASK_INDEX\"},
               {\"name\": \"TASK_COUNT\", \"value\": \"$TASK_COUNT\"},
               {\"name\": \"TIMESTAMP\", \"value\": \"$TIMESTAMP\"}
             ]
           }
         ]
       }" \
       --network-configuration "awsvpcConfiguration={
         subnets=[$SUBNET],
```

```
            securityGroups=[$SG],
            assignPublicIp=ENABLED
          }")

    TASK_ARN=$(echo "$OUT" | jq -r '.tasks[0].taskArn // empty')

    if [[ -n "$TASK_ARN" ]]; then
        echo "task $TASK_INDEX Accepted: $TASK_ARN"
        TASK_ARNS+=("$TASK_ARN")
        return 0
    else
        echo "task $TASK_INDEX Rejected: $(echo "$OUT" | jq -r
'.failures')"
        return 1
    fi
}

echo "Starting launch of $TASK_COUNT tasks..."

for ((i=0; i<TASK_COUNT; i++)); do
    while true; do
        if launch_task "$i"; then
            break
        else
            echo "Retrying task $i in 1s..."
            sleep 1
        fi
    done
    sleep 0.2
done

echo "All tasks submitted successfully!"

echo "TASK_ARNS=${TASK_ARNS[*]}"
```

Appendix 6: ECS Container Script Sequence Diagram

The below sequence diagram details the logic for a modification of an existing bash script in order to run Sikraken against a given set of benchmarks. The main modifications are that it only works as a single threaded process, as each ECS task is only given one vCPU, it also uses the variables passed down from the run_ecs_task.sh script such as the task ID, to ensure that each task processes a unique benchmark as well as the other variables passed down to configure Sikraken such as the budget.



Appendix 6.1: ECS Container Script

The container script allows for the individual ECS tasks to each process unique benchmarks from a given set so that the work may be spread across a given number of tasks such as 60. This is achieved by using a task ID (TASK_INDEX in code) and then sorting and placing all benchmark .yml files from a given set of categories into an array. The array is looped through for each index, and if the index modulo divided by the TASK_COUNT variable doesn't equate to the task ID, then the loop continues, if it does however equate to the task ID, then that benchmark with the corresponding .yml file is processed.

Each C benchmark has a corresponding .yml file with given information such as the data model to be used for each benchmark [79] which are taken into account before Sikraken processes the benchmark. Once Sikraken processes a benchmark, it then outputs its results to a directory in a shared volume in the container. The last main modification is that the Sikraken outputs are then outputted to the S3 bucket using the aws S3 sync command [100].

```
#!/bin/bash

#

# Script: test_category_sikraken_ecs.sh

# Author: Chris Meudec

# Modified By: Isaiah Andres

# Started: May 2025 (Major update: Oct 2025 for parallel error
handling)

# Modified On: Feb 2026
```

```
# Description: This is a modified version of an existing script
test_category_sikraken.sh to be single threaded so that it may work in
an ECS Fargate cluster.

#               Most features and options have been removed by
commenting them out as the focus for now is on measuring the
performance but have been kept

#               in case further development may be necessary, so
comments explaining other features have been removed until added again.

# The <category>.set file are in sv-benchmarks/c directory or can be
user-defined

# Outputs logs files into directory within a shared volume with a
docker container /shared/output

# Takes into account possible exclude set for ECA

# For each benchmark: generate tests, upload log to S3 Bucket


# Example: ./SikrakenDevSpace/bin/test_category_sikraken.sh
/home/chris/sv-benchmarks/c ECA 8 30 debug --ss=5


clear


echo "Starting Sikraken ECS run..."


echo "All benchmarks are present."


echo "Benchmarks content in /shared/benchmarks:"

ls /shared/benchmarks


S3_BUCKET_NAME="ecs-benchmarks-output"

S3_BUCKET="${S3_BUCKET_NAME:?S3_BUCKET not set}"

CORES="${CORES:-1}"
```

```
STACK_SIZE_GB="${STACK_SIZE_GB:-3}"

CATEGORY="${CATEGORY:-chris}"

MODE="${MODE:-release}"

BUDGET="${BUDGET:-10}"

TIMESTAMP="${TIMESTAMP:?TIMESTAMP environment variable not set}"


TASK_COUNT="${TASK_COUNT:-1}"

TASK_INDEX="${TASK_INDEX:-0}"

OUTPUT_SHARED="/shared/output"


SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"

SIKRAKEN_INSTALL_DIR="$SCRIPT_DIR/.."

BL='\033[34m'    # blue

YL="\033[38;5;226m"     # yellow

GR='\033[32m'    # green

RD='\033[31m'    # red

NC='\033[0m'     # reset

script_name=$(basename "$0")


echo "Run all the benchmarks from a TestComp category using
SIKRAKEN_INSTALL_DIR=$SIKRAKEN_INSTALL_DIR"


# --- Required arguments ---

path_to_benchmarks="/shared/benchmarks"

category=$CATEGORY

cores=1

budget=$BUDGET
```

```
mode=$MODE


# --- Initialize Optional Variables ---
shortcutgen=""
shortcutgen_flag=0
no_testcov=1 #Setting To 1 as no testcov usage yet in ECS version
branch_highlight=0
stack_size_gb=3


# --- Process Optional Arguments (Shift and Loop) ---
if [ $# -gt 0 ]; then
   shift 5
   while [ "$#" -gt 0 ]; do
       option="$1"
       case "$option" in
           "-scg")
               shortcutgen=", shortcut_gen"
               shortcutgen_flag=1
               ;;
           "-no_testcov")
               no_testcov=1
               ;;
           "-bh")
               branch_highlight=1
               ;;
           --ss=*)
               # 1. Extract the value after the '=' sign
```

```
                stack_size_gb="${1#*=}"
                ;;
            *)
                # Handle unknown options
                echo "Sikraken ERROR from $script_name: Unknown option: 
$option"
                echo "Usage: $script_name <path_to_benchmarks> 
<category> <cores> <budget> <mode> [OPTIONS]"
                echo "Options: [-scg] [-no_testcov] [-ss STACK_SIZE]"
                exit 1
                ;;
        esac
        shift
    done
fi

# --- Debug info ---
echo "path_to_benchmarks = $path_to_benchmarks"
echo "category           = $category"
echo "cores              = $cores"
echo "budget             = $budget"
echo "mode               = $mode"
echo "shortcutgen        = $shortcutgen"
echo "no_testcov         = $no_testcov"

check_benchmarks_path(){
    # Check if the path_to_benchmarks exists
    if [ ! -d "$path_to_benchmarks" ]; then
```

```
        echo "Sikraken ERROR from $script_name: the passed path to
category $path_to_benchmarks does not exist."
        exit 1
    fi
}
check_benchmarks_path


retrieve_category_file(){
    category_file="$category".set   #input file describing the category
    local_category_path="$SIKRAKEN_INSTALL_DIR/categories/"
    # 1. Check local directory (e.g. for non-Test-Comp sets such as
chris.set)
    full_path_to_category_file="$local_category_path/$category_file"
    if [ -f "$full_path_to_category_file" ]; then
        echo "Using local category file: $full_path_to_category_file"
    # 2. Check path from argument instead
    elif [ -f "$path_to_benchmarks/$category_file" ]; then
        full_path_to_category_file="$path_to_benchmarks/$category_file"
        echo "Using argument path category file:
$full_path_to_category_file"
    # 3. File not found in either location
    else
        echo "Sikraken ERROR from $script_name: The category file
'$category_file' was not found in either"
        echo "  - Local Path: $local_category_path"
        echo "  - Argument Path: $path_to_benchmarks"
        exit 1
    fi
```

```
    # Define exclusion set for ECA
    exclude_set=""
    if [ $category == "ECA" ]; then
        exclude_set="$SCRIPT_DIR/../ECA-excludes.set"    # local copy, actual exclude file (only category for which there is one) is at https://gitlab.com/sosy-lab/test-comp/bench-defs/-/tree/testcomp25/benchmark-defs/excludes?ref_type=tags
        if [ ! -f "$exclude_set" ]; then
            echo "Sikraken ERROR from $script_name: Exclusion set $exclude_set does not exist."
            exit 1
        fi
        echo "Sikraken $script_name log: Using the exclude set: "$exclude_set""
    fi

    echo "Sikraken $script_name log: called: $script_name $@"
}
retrieve_category_file

compile_parser(){
    # re-compile the parser in case it changed during development
    $SIKRAKEN_INSTALL_DIR/bin/compile_parser.sh
    if [ $? -ne 0 ]; then
        echo "Sikraken ERROR from $script_name: ERROR: Sikraken parser recompilation failed"
        exit 1
    else
```

```
       echo "Sikraken $script_name log: Sikraken parser successfully
recompiled"

   fi

}

compile_parser


set_output_directory(){

   output_dir="$OUTPUT_SHARED/$TIMESTAMP"

   echo "The output dir is $output_dir"

   mkdir -p "$output_dir"

}

set_output_directory

# function: generate_tests runs single threaded for ECS

# and terminated with 'return 1' instead of 'exit 1'.

generate_tests() {

   local benchmark="$1"

   local gcc_flag="$2"

   local testcov_data_model="$3"


   # Extract the basename of the file (without the path nor extension)

   local basename=$(basename "$benchmark")

   basename="${basename%.*}"

   local benchmark_output_dir="$output_dir"/"$basename"

   mkdir -p "$benchmark_output_dir"

   local sikraken_log="$benchmark_output_dir/sikraken.log"


   # Generate test inputs
```

```
    local benchmark_relative_path=$(realpath
--relative-to="$SIKRAKEN_INSTALL_DIR" "$benchmark")

    local sikraken_call="$SIKRAKEN_INSTALL_DIR/bin/sikraken.sh $mode
$gcc_flag budget[$budget] --ss=$stack_size_gb $benchmark_relative_path"

    echo -e "${BL}Calling Sikraken using: $sikraken_call${NC}"

    $sikraken_call >> "$sikraken_log" 2>&1

    ret_code=$?

    if [ $ret_code -ne 0 ]; then

        error="Sikraken ERROR from $script_name: error code $ret_code
for $basename, Call to Sikraken $sikraken_call failed"

        echo "$error" >> "$sikraken_log"

        echo -e "${RD}$error${NC}"

    else

        echo -e "${GR}Sikraken $script_name log: Test inputs generated
for $basename using $sikraken_call${NC}"

    fi



    if [[ "$mode" == "debug" ]]; then    #generate graph of timings


$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/helper/create_runtime_graph.
sh "$sikraken_log"

    fi



    if (( branch_highlight == 1 )); then     #generate highlighted HTML C
code with missing coverage


$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/helper/highlight_branches.sh
"$sikraken_log"
"$SIKRAKEN_INSTALL_DIR/sikraken_output/$basename/$basename.pl"
"$benchmark_output_dir/$basename.html"

    else
```

```
        echo -e "${YL}Skipping coverage branches highlighting${NC}"
    fi

    if (( no_testcov == 1 )); then
        echo -e "${YL}Skipping TestCov: relying on Sikraken coverage${NC}"
    else
        testcov_call="$SIKRAKEN_INSTALL_DIR/bin/run_testcov.sh"    # program
        testcov_args=( "$benchmark" "$testcov_data_model" )       # args as array
        echo -e "${BL}Calling Testcov using: $testcov_call ${testcov_args[*]}${NC}"

        # run it without eval, preserving arguments and quoting
        "$testcov_call" "${testcov_args[@]}" >"$benchmark_output_dir/testcov_call.log" 2>&1

        echo -e "${GR}Ended TestCov for $basename${NC}"
    fi
}

### MAIN starts here
start_wall_time=$(date +"%Y-%m-%d %H:%M:%S")     # Capture human-readable time and Unix timestamp for start
start_ts=$(date +%s)

category_extracted_benchmarks_files="$output_dir"/benchmark_files.txt #output list of benchmarks for the category
```

```
log_file="$output_dir"/category_test_run.log


#printf -v orig_cmd '%q ' "${ORIG_ARGV[@]}"

echo "Command Used to Generate the Category Test run: ${orig_cmd% }" >>
"$log_file"

echo "Timestamp: $TIMESTAMP" >> $log_file

echo "Category: $category" >> $log_file

echo "Mode: $mode" >> $log_file

echo "Budget: $budget" >> $log_file

echo "Cores: $cores" >> $log_file

echo "Options: shortcutgen: $shortcutgen_flag, no_testcov: $no_testcov"
>> $log_file


run_benchmark(){

   mapfile -t PATTERNS < <(

       grep -o '.*\/.*\.yml' "$full_path_to_category_file" | sort

   )


   if (( TASK_INDEX >= TASK_COUNT )); then

       echo "ERROR: TASK_INDEX ($TASK_INDEX) >= TASK_COUNT
($TASK_COUNT)"

       exit 1

   fi


   for i in "${!PATTERNS[@]}"; do

       if (( i % TASK_COUNT != TASK_INDEX )); then

           continue

       fi
```

```
        pattern_benchmark_directory="${PATTERNS[$i]}"


        for yml_file in
"$path_to_benchmarks"/$pattern_benchmark_directory; do
            # Exclude files listed in the exclusion set
            if [ -n "$exclude_set" ] && grep -Fxq "$yml_file"
"$exclude_set"; then
                # echo "Sikraken $script_name log: skipping excluded
file: $yml_file"
                continue
            fi
            echo "yml_file is $yml_file"
            if [ -f "$yml_file" ]; then
                # Check if the file contains the required property_file
line
                if grep -qE "^\s*- property_file:
\.\./properties/coverage-branches\.prp$" "$yml_file"; then
                    echo -e "${YL}Sikraken $script_name log: extracting
benchmark file from $yml_file${NC}"


                    # Extract the input file (match "input_files:
<filename>" for .c or .i files)
                    benchmark=$(grep "input_files:" "$yml_file" | sed -n
"s/^[[:space:]]*input_files:[[:space:]]*\(['\"]\?\)\(.*\)\1/\2/p")


                    # Extract the data model
                    data_model=$(grep "data_model:" "$yml_file" | sed -n
"s/^[[:space:]]*data_model:[[:space:]]*\(.*\)/\1/p")
```

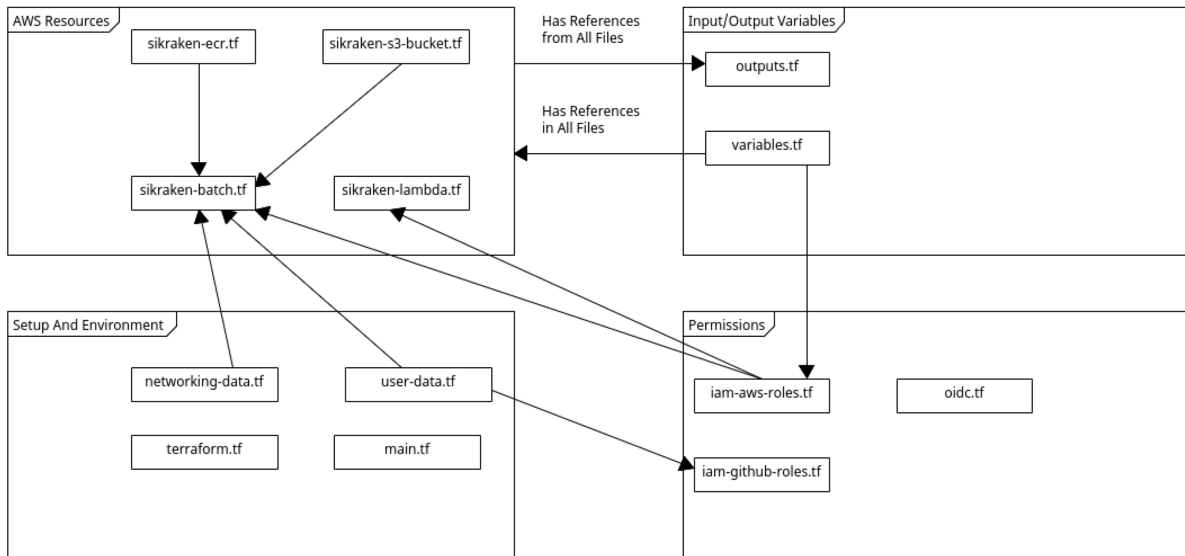
```
                    # Generate GCC flag based on the value of data_model

                    if [ "$data_model" == "ILP32" ]; then

                        gcc_flag="-m32"

                        testcov_data_model="-32"

                    elif [ "$data_model" == "LP64" ]; then

                        gcc_flag="-m64"

                        testcov_data_model="-64"

                    else

                        echo "Sikraken ERROR from $script_name:
unsupported data model: $data_model"

                        exit 1

                    fi

                    full_path_benchmark_file="$(dirname
"$yml_file")/$benchmark"

                    # write each file in the benchmark category into
$category_extracted_benchmarks_files used for table generation

                    echo "$full_path_benchmark_file $testcov_data_model"
>> $category_extracted_benchmarks_files


                    generate_tests "$full_path_benchmark_file"
"$gcc_flag" "$testcov_data_model"

                fi  # if the .yml file does not contain the correct
property, it is silently skipped

            fi

        done #no more *.yml file

    done


    # Capture human-readable time and Unix timestamp for end

    end_wall_time=$(date +"%Y-%m-%d %H:%M:%S")
```

```
    end_ts=$(date +%s)

    echo "Sikraken $script_name: Start Wall Time: $start_wall_time"

    echo "Sikraken $script_name: End Wall Time: $end_wall_time"

    duration_seconds=$((end_ts - start_ts))

    duration_hms=$(date -u -d @"$duration_seconds" +"%H:%M:%S")

    echo "Sikraken $script_name: Duration: $duration_hms"

    echo "Duration: $duration_hms" >> $log_file

}


run_benchmark

#generate_table_script="$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/help
er/create_category_test_run_table.sh $output_dir"

#echo "Sikraken $script_name: now calling $generate_table_script"

#$generate_table_script


# update the overall category summary table for all the previous runs

#generate_summary="$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/helper/vi
ew_category_compare.sh ./SikrakenDevSpace/categories/$category/"

#echo "Sikraken $script_name log: now calling $generate_summary"

#$generate_summary


upload_benchmark_to_s3(){

    S3_PREFIX="s3://${S3_BUCKET}/${CATEGORY}/${TIMESTAMP}"

    echo "$S3_PREFIX"

    aws s3 sync "$output_dir" "$S3_PREFIX" --exclude "*.i" --exclude
"*.log"


    # Upload .i and .log files with text/plain content-type
```

```
    aws s3 sync "$output_dir" "$S3_PREFIX" \
        --exclude "*" \
        --include "*.i" \
        --include "*.log" \
        --content-type text/plain

    echo "Sikraken $script_name log: has ended."
}
upload_benchmark_to_s3
```

Appendix 7: Terraform Diagram

The diagram below shows the different files and connections between them within the Terraform configuration and separated into different blocks based on their role. Explanations for each file and the block they've been contained in within the diagram are available after the diagram.



AWS Resources

The AWS Resources block includes the files responsible for creating the actual resources in AWS such as Batch, S3 buckets, ECR and the Lambda.

sikraken-batch.tf

The file sikraken-batch.tf is responsible for creating the AWS Batch resource as well as the other resources that it would depend on, such as the compute environment, the job queue and the job

definition. The file uses the Batch module provided through the Terraform Registry for creating the components. [156]

sikraken-ecr.tf

The file sikraken-ecr.tf simply creates the Elastic Container Registry with the name being "(project prefix)-images". The code shows the usage of a lifecycle policy in which untagged images are removed after one day. This is because when pushing an image with the same image tag to ECR seems to create several images with no tags, and will therefore take up more space and increase the cost of the ECR which creates the need for their removal to save costs. The ECR module on the Terraform Registry is used to create the resource. [157]

sikraken-lambda.tf

The file sikraken-lambda.tf creates the Lambda function with the name "(project prefix)-outputs-processor" and points to the output_report_url.py script in the lambda-functions directory. The code is dependent on the Lambda module from the Terraform Registry to create the resource. [158]

sikraken-s3-bucket.tf

The file sikraken-s3-bucket.tf is responsible for creating the S3 Buckets that store the outputs from the Sikraken test runs and also the TestComp benchmarks and is made using the S3 module from the Terraform Registry. [159]

Input/Output Variables

The Input/Output Variables block include files that take input from the user and uses them as variables throughout the configuration as well as output important data that the user will need to configure the GitHub Actions pipeline

outputs.tf

The file outputs.tf take outputs from the files in the AWS Resources and also iam-aws-roles.tf in order for the user to reference them in the GitHub Actions pipeline. These outputs, when referenced in the pipeline, point the pipeline to the resources created by the Terraform module through the use of the pipeline's environment variables such as the S3 bucket name, Lambda function name etc.

variables.tf

The file variables.tf take inputs from the user and then use them to decide variables such as unique resource names for resources that require them such as for S3 buckets, default settings and prefixes to attach to resource names that don't need unique names.

Setup And Environment

The Setup And Environment block contains files that define global options in the configuration such as the cloud service provider and networking details.

networking-data.tf

This file holds variables pointing to the default options for networking resources such as security groups, virtual private cloud (VPC) and subnets to be used across the configuration such as in the AWS Batch file.

user-data.tf

The user-data.tf file holds the account ID of the user and the region they belong to.

terraform.tf

The file terraform.tf state the required Terraform provider and their version for the Terraform configuration

main.tf

The file main.tf simply declares the provider and the region it belongs to.

Permissions

The Permissions block contains files that set up permissions throughout the configuration, such as for allowing other resources to interact with each other or for the pipeline to be able to interact with the resources generated by the configuration.

iam-aws-roles.tf

This file declares the IAM roles and permissions policies of each role for the Lambda and Batch services generated by the Terraform configuration. The IAM roles and IAM role policies were set up through resources from the Terraform Registry. [160, 161]

iam-github-roles.tf

This file holds the IAM roles and policies taken up within the pipeline that allows it to interact with the resources generated by the configuration such as the ECR, to push Docker images created in the pipeline to it, Batch to submit a job from the pipeline and Lambda so that it can be invoked from the pipeline. Similarly, the IAM roles and IAM role policies were set up through the resources provided from the Terraform Registry. [160, 161]

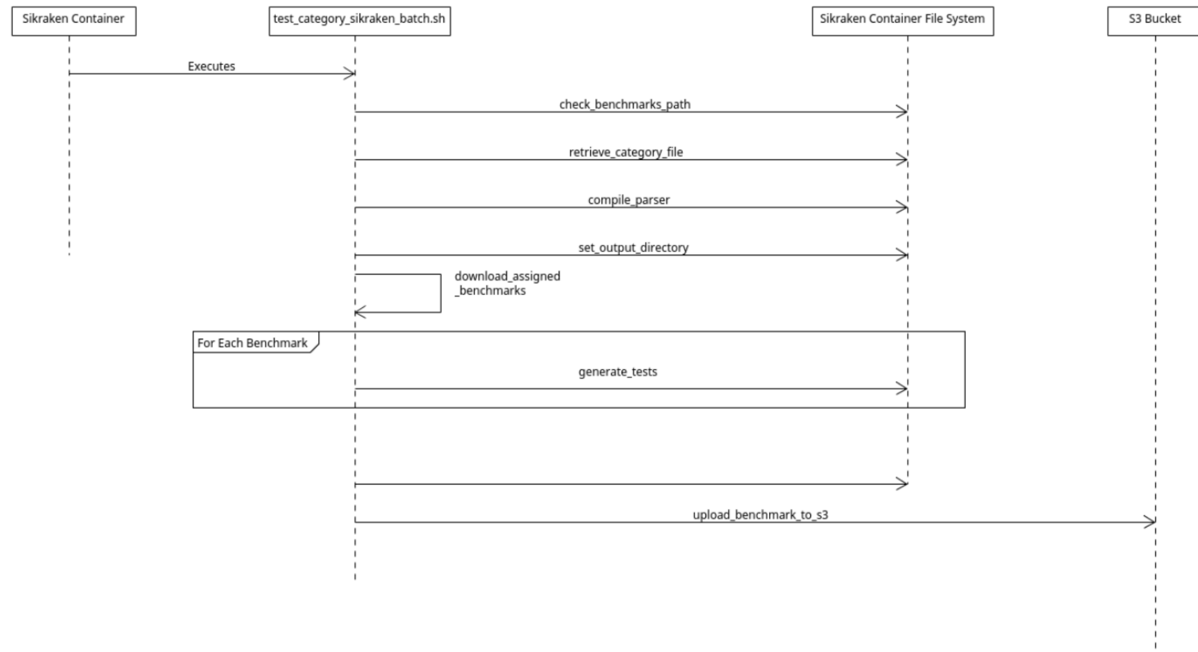
oidc.tf

The oidc.tf file is used in order to set up OpenID Connect (OIDC) with the user's AWS account and GitHub, allowing for the user to authenticate GitHub actions with their AWS account without needing to store any of their own credentials in the pipeline as a secret environment variable, allowing for better security. OIDC is set up in this file through the use of the AWS IAM Terraform module [162]

Appendix 8: Batch Container Script - Sequence Diagram

The sequence diagram below shows the script run inside the Sikraken container in order to start a test run. The script itself is only slightly different to the ECS implementation in the sense that it's a modification of an existing script, test_category_sikraken.sh but adapted for an AWS Batch

environment. The main difference is that it now includes .i files for each benchmark to be copied to their respective directories and the benchmarks are now downloaded within the container rather than having a dedicated benchmarks container upload its benchmarks to a volume shared between itself and the Sikraken container.



Appendix 8.1 Batch Container Script

Similarly to the ECS container script, the batch container script allows for the individual Batch tasks to each process unique benchmarks from a given set so that the work may be spread across a given number of jobs such as 60. The main difference is that the script now assigns all the names of all .yml files listed in the .set file for a category to an array to get the total number of benchmarks to download and process. This allows for the same algorithm to split the work of downloading and processing the benchmarks by looping through the array and modulo dividing each index by the total number of jobs and only downloading then processing the benchmark if the result of the calculation is equal to the job ID.

Downloading the benchmark that satisfies the parallelism algorithm involves taking the relative path of the .yml file that's stored in the .set file, creating directories based on it (the format is parent directory/benchmark name in the .set) and then downloading the benchmark from the S3 bucket to the newly created directory. The relative path of the benchmark assigned to the job is also appended to a new array which is used to construct a path to the benchmark in order for Sikraken to process it.

Processing each benchmark uses the same series of steps as the one seen in the ECS implementation in which data in the corresponding .yml file for each benchmark is retrieved such as the data model to be used before Sikraken processes it and generates its outputs.

A new addition to the script in comparison to the ECS script is that the .i files outputted by Sikraken are now retrieved by taking the name of the directory of the benchmark, and searching for a directory with the .i file with the same name as the benchmark it corresponds to in the directory of Sikraken's outputs. Following this, the data outputted by Sikraken is uploaded to the user's S3 bucket in the same way as the ECS script. The existing script, create_category_test_run_table.sh relies on .txt files generated by this script known as benchmarks.txt that contains a list of the benchmarks that were processed and their data model. Since this script is now single threaded and doesn't process all

benchmarks at once, benchmarks.txt now has the index of the array job running the script appended to the name before being uploaded to the S3. This way a later script can combine them after retrieving them all from the S3 bucket and allow for create_category_test_run_table.sh to function normally.

```
#!/bin/bash
#
# Script: test_category_sikraken_ecs.sh
# Author: Chris Meudec
# Modified By: Isaiah Andres
# Started: May 2025 (Major update: Oct 2025 for parallel error
handling)
# Modified On: March 2026
# Description: This is a modified version of an existing script
test_category_sikraken.sh to be single threaded so that it may work in
an AWS Batch job.
#              Most features and options have been removed by
commenting them out as the focus for now is on measuring the
performance but have been kept
#              in case further development may be necessary, so
comments explaining other features have been removed until added again.
# The <category>.set file are in sv-benchmarks/c directory or can be
user-defined
# Outputs logs files into directory within a shared volume with a
docker container /shared/output
# Takes into account possible exclude set for ECA
# For each benchmark: generate tests, upload log to S3 Bucket

# Example: ./SikrakenDevSpace/bin/test_category_sikraken.sh
/home/chris/sv-benchmarks/c ECA 8 30 debug --ss=5

clear

echo "Starting Sikraken Batch run..."
mkdir -p "/benchmarks"

S3_BUCKET_NAME="${S3_BUCKET_NAME:-ecs-benchmarks-output}"
S3_BUCKET="${S3_BUCKET_NAME:?S3_BUCKET not set}"
TESTCOMP_S3_BUCKET_NAME="${TESTCOMP_S3_BUCKET_NAME:-testcomp-benchmarks
}"
TESTCOMP_BUCKET="${TESTCOMP_S3_BUCKET_NAME:?TESTCOMP_S3_BUCKET_NAME not
set}"
CORES="${CORES:-1}"
STACK_SIZE_GB="${STACK_SIZE_GB:-3}"
CATEGORY="${CATEGORY:-chris}"
```

```
MODE="${MODE:-release}"
BUDGET="${BUDGET:-10}"
BRANCH_HIGHLIGHTING="${BRANCH_HIGHLIGHTING:-0}"
TIMESTAMP="${TIMESTAMP:?TIMESTAMP environment variable not set}"

#Using Batch environment variables
JOB_COUNT="${JOB_COUNT:-1}"
JOB_INDEX="${AWS_BATCH_JOB_ARRAY_INDEX:-0}"
OUTPUT_SHARED="/output"

SCRIPT_DIR="$(cd "$(dirname "$0")" && pwd)"
SIKRAKEN_INSTALL_DIR="$SCRIPT_DIR/.."
BL='\033[34m'     # blue
YL="\033[38;5;226m"     # yellow
GR='\033[32m'     # green
RD='\033[31m'     # red
NC='\033[0m'      # reset
script_name=$(basename "$0")

echo "Run all the benchmarks from a TestComp category using
SIKRAKEN_INSTALL_DIR=$SIKRAKEN_INSTALL_DIR"

# --- Required arguments ---
path_to_benchmarks="/benchmarks"
category=$CATEGORY
cores=1
budget=$BUDGET
mode=$MODE

# --- Initialize Optional Variables ---
shortcutgen=""
shortcutgen_flag=0
no_testcov=1 #Setting To 1 as no testcov usage yet in Batch version
branch_highlight=$BRANCH_HIGHLIGHTING
stack_size_gb=$STACK_SIZE_GB

# --- Process Optional Arguments (Shift and Loop) ---
if [ $# -gt 0 ]; then
   shift 5
   while [ "$#" -gt 0 ]; do
       option="$1"
       case "$option" in
           "-scg")
```

```
                shortcutgen=", shortcut_gen"
                shortcutgen_flag=1
                ;;
            *)
                # Handle unknown options
                echo "Sikraken ERROR from $script_name: Unknown option: 
$option"
                echo "Usage: $script_name <path_to_benchmarks> 
<category> <cores> <budget> <mode> [OPTIONS]"
                echo "Options: [-scg] [-no_testcov] [-ss STACK_SIZE]"
                exit 1
                ;;
        esac
        shift
    done
fi

# --- Debug info ---
echo "path_to_benchmarks = $path_to_benchmarks"
echo "category           = $category"
echo "budget             = $budget"
echo "mode               = $mode"
echo "shortcutgen        = $shortcutgen"
echo "job_index         = $JOB_INDEX"
echo "job_count         = $JOB_COUNT"
echo "stack_size         = $stack_size_gb"

check_benchmarks_path(){
    # Check if the path_to_benchmarks exists
    if [ ! -d "$path_to_benchmarks" ]; then
        echo "Sikraken ERROR from $script_name: the passed path to 
category $path_to_benchmarks does not exist."
        exit 1
    fi
}
check_benchmarks_path

retrieve_category_file(){
    category_file="$category".set   #input file describing the category
    local_category_path="$SIKRAKEN_INSTALL_DIR/categories/"
    # 1. Check local directory (e.g. for non-Test-Comp sets such as 
chris.set)
    full_path_to_category_file="$local_category_path/$category_file"
```

```
   if [ -f "$full_path_to_category_file" ]; then
       echo "Using local category file: $full_path_to_category_file"
   # 2. Check path from argument instead
   elif [ -f "$path_to_benchmarks/$category_file" ]; then
       full_path_to_category_file="$path_to_benchmarks/$category_file"
       echo "Using argument path category file:
$full_path_to_category_file"
   # 3. File not found in either location
   else
       echo "Sikraken ERROR from $script_name: The category file
'$category_file' was not found in either"
       echo "  - Local Path: $local_category_path"
       echo "  - Argument Path: $path_to_benchmarks"
       exit 1
   fi

   # Define exclusion set for ECA
   exclude_set=""
   if [ $category == "ECA" ]; then
       exclude_set="$SCRIPT_DIR/../ECA-excludes.set"   # local copy,
actual exclude file (only category for which there is one) is at
https://gitlab.com/sosy-lab/test-comp/bench-defs/-/tree/testcomp25/benc
hmark-defs/excludes?ref_type=tags
       if [ ! -f "$exclude_set" ]; then
           echo "Sikraken ERROR from $script_name: Exclusion set
$exclude_set does not exist."
           exit 1
       fi
       echo "Sikraken $script_name log: Using the exclude set:
"$exclude_set""
   fi

   echo "Sikraken $script_name log: called: $script_name $@"
}
retrieve_category_file

compile_parser(){
   # re-compile the parser in case it changed during development
   $SIKRAKEN_INSTALL_DIR/bin/compile_parser.sh
   if [ $? -ne 0 ]; then
       echo "Sikraken ERROR from $script_name: ERROR: Sikraken parser
recompilation failed"
       exit 1
```

```
    else
        echo "Sikraken $script_name log: Sikraken parser successfully
recompiled"
    fi
}
compile_parser

set_output_directory(){
    output_dir="$OUTPUT_SHARED/$TIMESTAMP"
    echo "The output dir is $output_dir"
    mkdir -p "$output_dir"
}
set_output_directory
# function: generate_tests runs single threaded for ECS
# and terminated with 'return 1' instead of 'exit 1'.
generate_tests() {
    local benchmark="$1"
    local gcc_flag="$2"
    local testcov_data_model="$3"

    # Extract the basename of the file (without the path nor extension)
    local basename=$(basename "$benchmark")
    basename="${basename%.*}"
    local benchmark_output_dir="$output_dir"/"$basename"
    mkdir -p "$benchmark_output_dir"
    local sikraken_log="$benchmark_output_dir/sikraken.log"

    # Generate test inputs
    local benchmark_relative_path=$(realpath
--relative-to="$SIKRAKEN_INSTALL_DIR" "$benchmark")
    local sikraken_call="$SIKRAKEN_INSTALL_DIR/bin/sikraken.sh $mode
$gcc_flag budget[$budget] --ss=$stack_size_gb $benchmark_relative_path"
    echo -e "${BL}Calling Sikraken using: $sikraken_call${NC}"
    $sikraken_call >> "$sikraken_log" 2>&1
    ret_code=$?
    if [ $ret_code -ne 0 ]; then
        error="Sikraken ERROR from $script_name: error code $ret_code
for $basename, Call to Sikraken $sikraken_call failed"
        echo "$error" >> "$sikraken_log"
        echo -e "${RD}$error${NC}"
    else
        echo -e "${GR}Sikraken $script_name log: Test inputs generated
for $basename using $sikraken_call${NC}"
```

```
   fi

   if [[ "$mode" == "debug" ]]; then    #generate graph of timings

$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/helper/create_runtime_graph.
sh "$sikraken_log"
   fi

   if (( branch_highlight == 1 )); then     #generate highlighted HTML C
code with missing coverage

$SIKRAKEN_INSTALL_DIR/SikrakenDevSpace/bin/helper/highlight_branches.sh
"$sikraken_log"
"$SIKRAKEN_INSTALL_DIR/sikraken_output/$basename/$basename.pl"
"$benchmark_output_dir/$basename.html"
   else
       echo -e "${YL}Skipping coverage branches highlighting${NC}"
   fi

   if (( no_testcov == 1 )); then      #No TestCov support for Batch,
removed debug message for it but keeping this line
       echo -e "${YL}Skipping TestCov: relying on Sikraken
coverage${NC}"
   else
       testcov_call="$SIKRAKEN_INSTALL_DIR/bin/run_testcov.sh"    #
program
       testcov_args=( "$benchmark" "$testcov_data_model" )       # args
as array
       echo -e "${BL}Calling Testcov using: $testcov_call
${testcov_args[*]}${NC}"

       # run it without eval, preserving arguments and quoting
       "$testcov_call" "${testcov_args[@]}"
>"$benchmark_output_dir/testcov_call.log" 2>&1

       echo -e "${GR}Ended TestCov for $basename${NC}"
   fi
}

### MAIN starts here
start_wall_time=$(date +"%Y-%m-%d %H:%M:%S")     # Capture
human-readable time and Unix timestamp for start
start_ts=$(date +%s)
```

```
mkdir -p "$output_dir/benchmark_files"
category_extracted_benchmarks_files="$output_dir"/benchmark_files/bench
mark_files-$JOB_INDEX.txt  #output list of benchmarks for the category
log_file="$output_dir"/category_test_run.log

#printf -v orig_cmd '%q ' "${ORIG_ARGV[@]}"
echo "Command Used to Generate the Category Test run: ${orig_cmd% }" >>
"$log_file"
echo "Timestamp: $TIMESTAMP" >> $log_file
echo "Category: $category" >> $log_file
echo "Mode: $mode" >> $log_file
echo "Budget: $budget" >> $log_file
echo "Cores: $cores" >> $log_file
echo "Options: shortcutgen: $shortcutgen_flag, no_testcov: $no_testcov"
>> $log_file

retrieve_all_yml_files() {
   mapfile -t PATTERNS < <(
       grep -o '.*\/.*\.yml' "$full_path_to_category_file" | sort
   )
}

retrieve_all_yml_files

download_assigned_benchmarks() {
   TESTCOMP_BUCKET_PREFIX="c"

   mkdir -p "$path_to_benchmarks"

   for i in "${!PATTERNS[@]}"; do
       if (( i % JOB_COUNT != JOB_INDEX )); then
           continue
       fi

       rel_path="${PATTERNS[$i]}"
       dir_name="$(dirname "$rel_path")"
       local_yml_path="$path_to_benchmarks/$rel_path"

       echo "Downloading YAML: $rel_path"

       mkdir -p "$path_to_benchmarks/$dir_name"

       aws s3 cp \
```

```
            "s3://$TESTCOMP_BUCKET/$TESTCOMP_BUCKET_PREFIX/$rel_path" \
            "$local_yml_path"

        if [ ! -f "$local_yml_path" ]; then
            echo "Sikraken ERROR: Failed to download $rel_path"
            continue
        fi

        benchmark_file=$(grep "input_files:" "$local_yml_path" \
            | sed -n
"s/^[[:space:]]*input_files:[[:space:]]*\(['\"]\?\)\(.*\)\1/\2/p")

        if [ -z "$benchmark_file" ]; then
            echo "Sikraken ERROR: Could not extract input_files from
$rel_path"
            continue
        fi

        echo "Downloading benchmark input file:
$dir_name/$benchmark_file"

        aws s3 cp \

"s3://$TESTCOMP_BUCKET/$TESTCOMP_BUCKET_PREFIX/$dir_name/$benchmark_fil
e" \
            "$path_to_benchmarks/$dir_name/$benchmark_file"

        if [ $? -ne 0 ]; then
            echo "Sikraken ERROR: Failed to download benchmark file
$benchmark_file"
            continue
        fi
    done
}
download_assigned_benchmarks

run_benchmark(){
    for i in "${!PATTERNS[@]}"; do
        if (( i % JOB_COUNT != JOB_INDEX )); then
            continue
        fi

        pattern_benchmark_directory="${PATTERNS[$i]}"
```

```
       for yml_file in
"$path_to_benchmarks"/$pattern_benchmark_directory; do
           # Exclude files listed in the exclusion set
           if [ -n "$exclude_set" ] && grep -Fxq "$yml_file"
"$exclude_set"; then
               # echo "Sikraken $script_name log: skipping excluded
file: $yml_file"
               continue
           fi
           echo "yml_file is $yml_file"
           if [ -f "$yml_file" ]; then
               # Check if the file contains the required property_file
line
               if grep -qE "^\s*- property_file:
\.\./properties/coverage-branches\.prp$" "$yml_file"; then
                   echo -e "${YL}Sikraken $script_name log: extracting
benchmark file from $yml_file${NC}"

                   # Extract the input file (match "input_files:
<filename>" for .c or .i files)
                   benchmark=$(grep "input_files:" "$yml_file" | sed -n
"s/^[[:space:]]*input_files:[[:space:]]*\(['\"]\?\)\(.*\)\1/\2/p")

                   # Extract the data model
                   data_model=$(grep "data_model:" "$yml_file" | sed -n
"s/^[[:space:]]*data_model:[[:space:]]*\(.*\)/\1/p")

                   # Generate GCC flag based on the value of data_model
                   if [ "$data_model" == "ILP32" ]; then
                       gcc_flag="-m32"
                       testcov_data_model="-32"
                   elif [ "$data_model" == "LP64" ]; then
                       gcc_flag="-m64"
                       testcov_data_model="-64"
                   else
                       echo "Sikraken ERROR from $script_name:
unsupported data model: $data_model"
                       exit 1
                   fi
                   full_path_benchmark_file="$(dirname
"$yml_file")/$benchmark"
```

```
                    # write each file in the benchmark category into
$category_extracted_benchmarks_files used for table generation
                    echo "$full_path_benchmark_file $testcov_data_model"
>> $category_extracted_benchmarks_files

                    generate_tests "$full_path_benchmark_file"
"$gcc_flag" "$testcov_data_model"
                fi  # if the .yml file does not contain the correct
property, it is silently skipped
            fi
        done #no more *.yml file
    done

    # Capture human-readable time and Unix timestamp for end
    end_wall_time=$(date +"%Y-%m-%d %H:%M:%S")
    end_ts=$(date +%s)
    echo "Sikraken $script_name: Start Wall Time: $start_wall_time"
    echo "Sikraken $script_name: End Wall Time: $end_wall_time"
    duration_seconds=$((end_ts - start_ts))
    duration_hms=$(date -u -d @"$duration_seconds" +"%H:%M:%S")
    echo "Sikraken $script_name: Duration: $duration_hms"
    echo "Duration: $duration_hms" >> $log_file
}

run_benchmark

copy_i_files_to_corresponding_folders(){
    echo "Copying .i files..."
    for d in "$output_dir"/*/; do
        name=$(basename "$d")
        src_file="$SIKRAKEN_INSTALL_DIR/sikraken_output/$name/$name.i"
        if [[ -f "$src_file" ]]; then
        cp "$src_file" "$d"
        echo "Copied $src_file → $d"
        else
        echo "Missing: $src_file"
        fi
    done
}
copy_i_files_to_corresponding_folders

upload_benchmark_to_s3(){
    S3_PREFIX="s3://${S3_BUCKET}/${CATEGORY}/${TIMESTAMP}"
```

```
    echo "$S3_PREFIX"
    aws s3 sync "$output_dir" "$S3_PREFIX" --exclude "*.i" --exclude
"*.log"

    # Upload .i and .log files with text/plain content-type
    aws s3 sync "$output_dir" "$S3_PREFIX" \
        --exclude "*" \
        --include "*.i" \
        --include "*.log" \
        --content-type text/plain

    echo "Sikraken $script_name log: has ended."
}
upload_benchmark_to_s3
```

Appendix 9: Run Batch Script

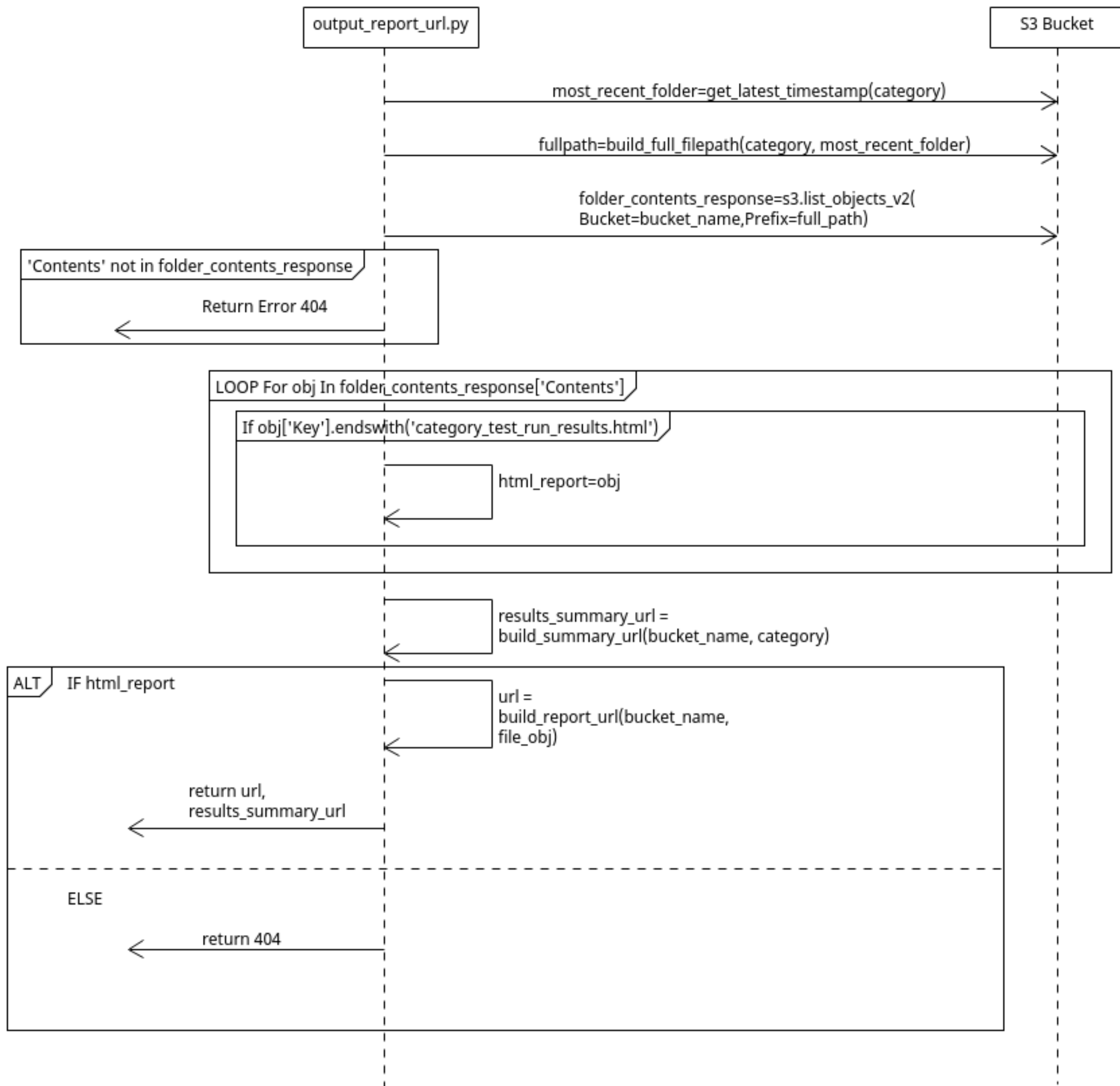
The script below shows the use of the command “aws batch submit-job” [155] to submit two jobs to be executed according to the given environment variables passed down from the pipeline. The job ID is outputted to the pipeline in order to give the pipeline the ability to get the status of the second job that will be outputted to the pipeline with the “describe-jobs” command, letting the pipeline know if it should keep waiting or continue to the next step depending on the status.

```
#!/usr/bin/env bash
set -euo pipefail

JOB_QUEUE="${1:-${JOB_QUEUE:-sikraken-test-comp-job-queue}}"
SIKRAKEN_JOB_DEFINITION="${2:-${SIKRAKEN_JOB_DEFINITION:-sikraken-test-
comp-batch-job-def}}"
JOB_COUNT="${3:-${JOB_COUNT:-5}}"
CATEGORY="${4:-${CATEGORY:-chris}}"
BUDGET="${5:-${BUDGET:-10}}"
MODE="${6:-${MODE:-release}}"
STACK_SIZE_GB="${7:-${STACK_SIZE_GB:-3}}"
S3_BUCKET_NAME="${8:-${S3_BUCKET_NAME:-ecs-benchmarks-output}}"
TESTCOMP_S3_BUCKET_NAME="${9:-${TESTCOMP_S3_BUCKET_NAME:-testcomp-bench
marks}}"
REPORT_JOB_DEFINITION="${10:-${REPORT_JOB_DEFINITION:-generate-report}}
"
BRANCH_HIGHLIGHTING="${11:-${BRANCH_HIGHLIGHTING:-0}}"
TIMESTAMP=$(date -u +"%Y_%m_%d_%H_%M")

JOB_ID=$(aws batch submit-job \
 --job-name "sikraken-${CATEGORY}-${TIMESTAMP}" \
```

```
 --job-queue "$JOB_QUEUE" \
 --job-definition "$SIKRAKEN_JOB_DEFINITION" \
 --array-properties size="$JOB_COUNT" \
 --retry-strategy '{"attempts": 5}' \
 --container-overrides "resourceRequirements=[
   {type=MEMORY,value=$(($STACK_SIZE_GB * 1024))}
 ],environment=[
   {name=CATEGORY,value=$CATEGORY},
   {name=BUDGET,value=$BUDGET},
   {name=MODE,value=$MODE},
   {name=TIMESTAMP,value=$TIMESTAMP},
   {name=STACK_SIZE_GB,value=$STACK_SIZE_GB},
   {name=JOB_COUNT,value=$JOB_COUNT},
   {name=S3_BUCKET_NAME,value=$S3_BUCKET_NAME},
   {name=TESTCOMP_S3_BUCKET_NAME,value=$TESTCOMP_S3_BUCKET_NAME},
   {name=BRANCH_HIGHLIGHTING,value=$BRANCH_HIGHLIGHTING}
 ]" \
 --query 'jobId' \
 --output text
)

JOB_ID2=$(aws batch submit-job \
 --job-name "generate-report-${CATEGORY}-${TIMESTAMP}" \
 --job-queue "$JOB_QUEUE" \
 --job-definition "$REPORT_JOB_DEFINITION" \
 --depends-on "[{\"jobId\": \"$JOB_ID\", \"type\": \"SEQUENTIAL\"}]" \
 --retry-strategy '{"attempts": 5}' \
 --container-overrides "environment=[
   {name=CATEGORY,value=$CATEGORY},
   {name=S3_BUCKET_NAME,value=$S3_BUCKET_NAME}
 ]" \
 --query 'jobId' \
 --output text
)

echo "$JOB_ID2"
```

Appendix 10: Lambda Script Sequence Diagram

The below sequence diagram details the script within the AWS Lambda to find the .html report and results summary and outputs it to the user.



Appendix 10.1: Lambda Script

The script below follows a very similar approach to the Lambda used in the EC2 implementation such that it takes the bucket name of the S3 bucket to search in. However this code has additional functionalities to also search for the results summary used and take JSON requests from the pipeline so the S3 bucket and the category that the benchmarks test run belongs to isn't hardcoded and is instead provided by the user.

The script takes advantage of the boto3 SDK's functions such as list_objects_v2, to find certain objects in the bucket such as the report generated by the test run based on a name and prefix as it returns a list of objects known as 'Contents', which contains metadata such as an object's 'Key' which is essentially a filename [164].

The name of the results summary is always results_summary.html and so has its url built and is at the base of the directory containing the benchmark category in the S3 bucket, so no searching is necessary. If the .html report is found then the url is constructed. Both urls are based on the S3 URI

format [148]. 404 Errors are returned if the html_report isn't found or if the contents of the latest timestamp is empty.

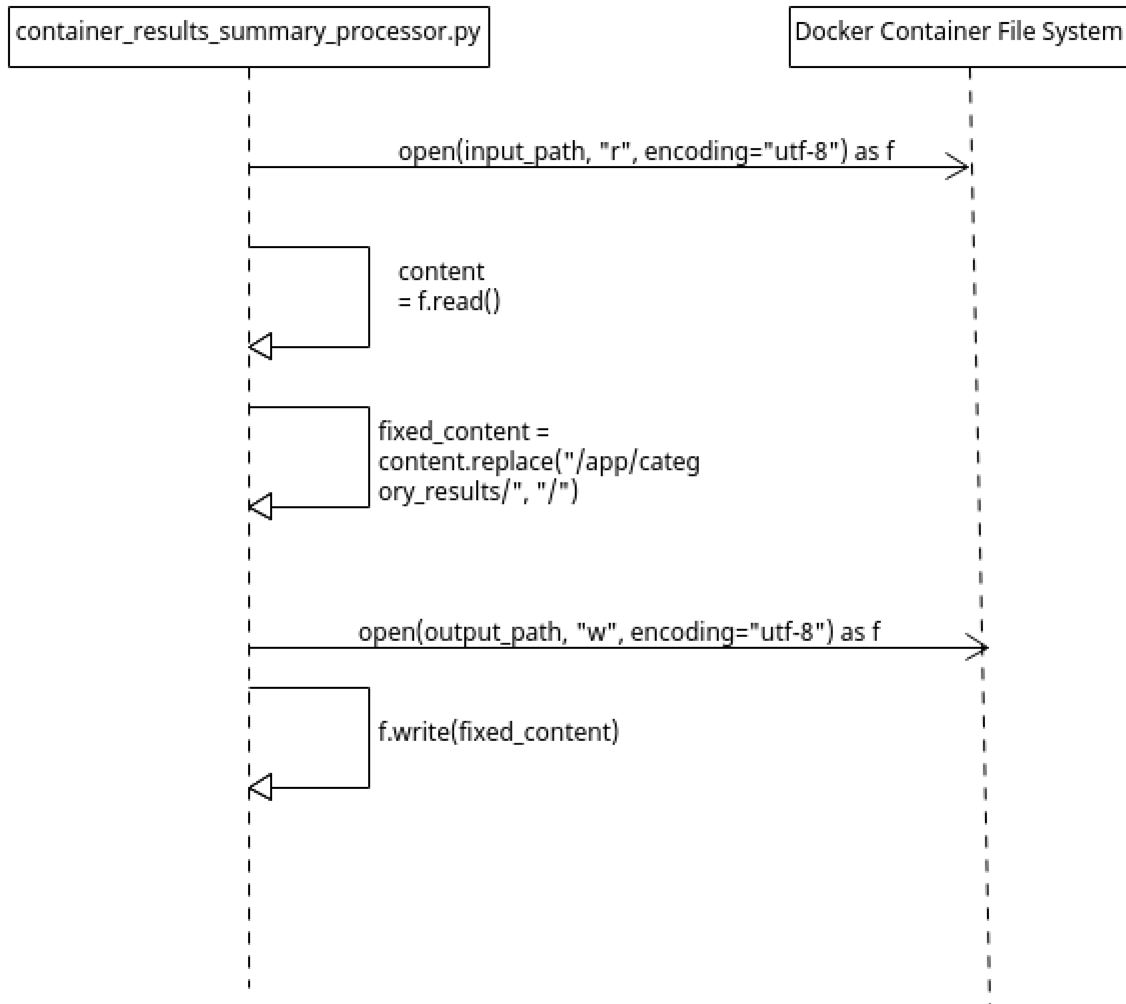
```
import json
import boto3

# Initialize S3 client
s3 = boto3.client('s3')
def retrieve_benchmarks(prefix, bucket_name):
   resp = s3.list_objects_v2(
       Bucket=bucket_name,
       Prefix=prefix,
       Delimiter="/"
   )
   return [p["Prefix"] for p in resp.get("CommonPrefixes", [])]

def get_latest_timestamp(category, bucket_name):
   prefixes = retrieve_benchmarks(f"{category}/", bucket_name)
   timestamp_filepaths = [p.rstrip("/").split("/")[-1] for p in
prefixes]
   timestamp_filepaths.sort(reverse=True)
   return timestamp_filepaths[0]

def build_full_filepath(category, most_recent_folder):
   return f"{category}/{most_recent_folder}"

def build_report_url(bucket_name, file_obj):
   return f"https://{bucket_name}.s3.amazonaws.com/{file_obj['Key']}"

def build_summary_url(bucket_name, category):
   return
f"https://{bucket_name}.s3.amazonaws.com/{category}/results_summary.htm
l"

def lambda_handler(event, context):
   bucket_name = event["Bucket"]
   object_key = 'category_test_run_results.html'  # File to search for
   category = event["Category"]

   most_recent_folder = get_latest_timestamp(category, bucket_name)

   full_path = build_full_filepath(category, most_recent_folder)
   #Listing the contents of the most recent folder found to find the
HTML file
```

```
    folder_contents_response = s3.list_objects_v2(
        Bucket=bucket_name,
        Prefix=full_path
    )

    if 'Contents' not in folder_contents_response:
        return {
            'statusCode': 404,
            'body': json.dumps(f'No contents found in folder
{most_recent_folder}.')
        }

    # Checking for object with the name of report generated
    html_report = [obj for obj in folder_contents_response['Contents']
if obj['Key'].endswith(object_key)]
    results_summary_url = build_summary_url(bucket_name, category)

    if html_report:
        file_obj = html_report[0] #Getting first S3 object found that
has the correct key name as a list is returned
        url = build_report_url(bucket_name, file_obj)

        return {
            'statusCode': 200,
            'body': json.dumps({
                'message': f"Found the HTML file: {file_obj['Key']}",
                'url': url,  #.html file url from S3 bucket that can be
viewed by anyone with the link
                'results_summary_url': results_summary_url
            })
        }
    else:
        return {
            'statusCode': 404, # Error message in the case that
            'body': json.dumps(f'{object_key} not found in folder
{most_recent_folder}.')
        }
```

Appendix 11: Container Results Summary Processor - Sequence Diagram

The sequence diagram below shows the logic to replace the docker container file paths that would be outputted to the .html results summary when linking the urls to past reports since executing the script

that generates it from the docker container will cause it to add the file path of the current working directory, “/app/category_results” to the urls of the reports.



Appendix 11.1: Container Results Summary Processor Script

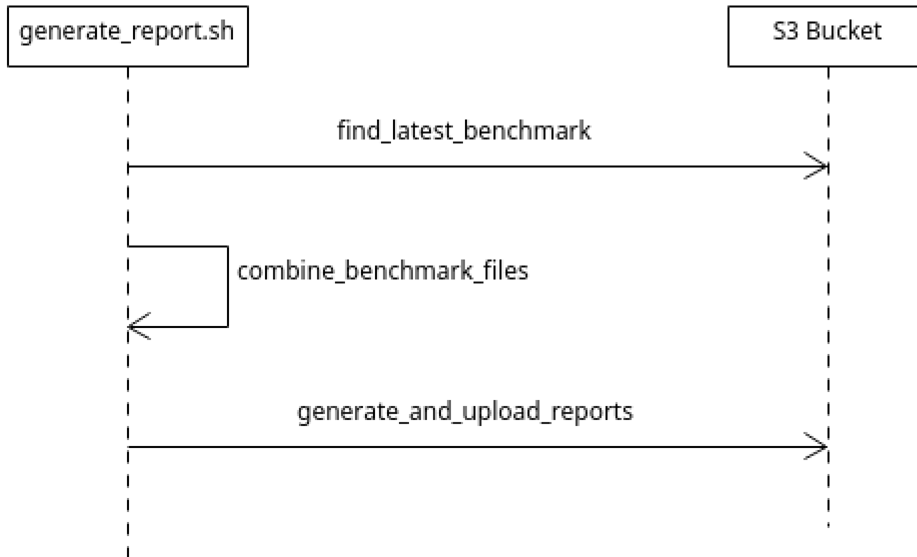
The script below simply reads a given file path and assigns the contents to a variable. The string “/app/category_results/” is then replaced with a single “/” and written back to an output file or the original content is overwritten with the new urls. This removes the string from the urls that link to the previous test run reports of a given category.

```
# Simple python script to fix urls in S3 as the full path rather than
relative is placed inside.

import sys
import re

def fix_s3_paths(input_path: str, output_path: str) -> None:
    PREFIX = "/app/category_results/"
```

```
    with open(input_path, "r", encoding="utf-8") as f:
        content = f.read()

    fixed_content = content.replace(PREFIX, "/")

    with open(output_path, "w", encoding="utf-8") as f:
        f.write(fixed_content)

    print(f"Output written to: {output_path}")


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python fix_s3_paths.py <input_file>
[output_file]")
        sys.exit(1)

    input_file = sys.argv[1]
    output_file = sys.argv[2] if len(sys.argv) > 2 else input_file  #
overwrite if no output specified

    fix_s3_paths(input_file, output_file)
```

Appendix 12: Report Container Script - Sequence Diagram

The sequence diagram below shows the series of functions called to generate reports that visualise the results from the Sikraken test run. The script is run inside a separate container that holds all the report scripts and is submitted as a job that's dependent on the Sikraken job.



Appendix 12.1: Report Container Script

The script below takes the user's S3 bucket and benchmark category as input before retrieving the latest benchmark for the category stored in the S3 bucket using regex. The benchmark.txt files with the job ID appended to the file name that were generated from the script executed in the Sikraken container are now combined into one benchmarks.txt file before the existing bash scripts for results visualisation are called. The outputs from each script are then processed and the outputs are uploaded to the user's S3 bucket.

```
#!/bin/bash

s3_bucket="${1:-${S3_BUCKET_NAME:-ecs-benchmarks-output}}"
CATEGORY="${2:-${CATEGORY:-ECA}}"

find_latest_benchmark(){
   aws s3 cp s3://$s3_bucket/$CATEGORY/ category_results/$CATEGORY
--recursive
   TIMESTAMP_DIR=$(find category_results/$CATEGORY -mindepth 1
-maxdepth 1 -type d -regextype posix-extended -regex
".*/[0-9]{4}_[0-9]{2}_[0-9]{2}_[0-9]{2}_[0-9]{2}" | sort | tail -n1)
}
find_latest_benchmark

combine_benchmark_files(){
   touch $TIMESTAMP_DIR/benchmark_files.txt
   cat $TIMESTAMP_DIR/benchmark_files/* >
$TIMESTAMP_DIR/benchmark_files.txt
}
```

```
combine_benchmark_files

generate_and_upload_reports(){
   /app/ReportScripts/create_category_test_run_table.sh
"$TIMESTAMP_DIR"

   TIMESTAMP_NAME=$(basename "$TIMESTAMP_DIR")
   python3 /app/SikrakenPythonScripts/filepath_to_url_processor.py
"$TIMESTAMP_DIR" --run_folder "$TIMESTAMP_NAME" --s3_bucket
"$s3_bucket" --category "$CATEGORY"
   aws s3 cp "$TIMESTAMP_DIR/category_test_run_results.html"
"s3://$s3_bucket/$CATEGORY/$TIMESTAMP_NAME/category_test_run_results.ht
ml" --content-type text/html

   /app/ReportScripts/view_category_compare.sh
category_results/$CATEGORY
   python3
/app/SikrakenPythonScripts/container_results_summary_processor.py
/app/category_results/$CATEGORY/results_summary.html
   aws s3 cp "/app/category_results/$CATEGORY/results_summary.html"
"s3://$s3_bucket/$CATEGORY/results_summary.html" --content-type
text/html
}
generate_and_upload_reports
```

Appendix 13: Sikraken Dockerfile

The dockerfile below shows the sequence of Docker commands used to create a Sikraken image. It first installs the necessary Ubuntu packages such wget, and ca-certificates [165] to install any necessary packages and trust the certificate authority [166] of whatever needs to be installed such as the AWS CLI. Flex and bison are requirements from Sikraken and removing the listed files in /var/lib/apt/lists is part of Docker's best practices [94] for building Docker images.

The Sikraken installation mainly involves the instructions listed in the SIkraken Install, Development and User guide with the main difference being that ECLiPSe Prolog's PATH is set up from the container and therefore the arguments –no-docs and –no-links are used to prevent prompts in the terminal.

Repositories such as Sikraken and directories that may come from repositories such as SikrakenDevSpace/categories that were checked out from the pipeline and added to its working directory are copied into the container's file system. Upon running the image that's built from the dockerfile, the script test_category_sikraken_batch.sh is executed.

```
FROM ubuntu:24.04
```

```
WORKDIR /app


RUN DEBIAN_FRONTEND=noninteractive apt-get update && apt-get install -y
\
   wget curl unzip gcc ca-certificates \
   flex bison \
   && rm -rf /var/lib/apt/lists/*


RUN curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
awscliv2.zip \
   && unzip awscliv2.zip \
   && ./aws/install \
   && rm -rf aws awscliv2.zip


COPY sikraken /app/sikraken
#COPY . /app/sikraken
COPY Batch/test_category_sikraken_batch.sh
/app/sikraken/bin/test_category_sikraken_batch.sh
COPY SikrakenDevSpace/bin/helper/highlight_branches.sh
/app/sikraken/SikrakenDevSpace/bin/helper/highlight_branches.sh
COPY SikrakenDevSpace/categories /app/sikraken/categories


WORKDIR /app/sikraken/eclipse
RUN wget
https://eclipseclp.org/Distribution/Builds/7.1_13/x86_64_linux/eclipse_
basic.tgz \
   && tar xzf eclipse_basic.tgz \
   && rm eclipse_basic.tgz \
   && chmod +x RUNME ARCH \
```

```
    && ./RUNME --no-docs --no-link


ENV ECLIPSEDIR=/app/sikraken/eclipse

ENV PATH="$ECLIPSEDIR/bin/x86_64_linux:$PATH"


RUN chmod +x /app/sikraken/bin/test_category_sikraken_batch.sh


WORKDIR /app/sikraken


ENTRYPOINT ["./bin/test_category_sikraken_batch.sh"]
```

Appendix 14: Reports Dockerfile

The dockerfile below builds a Docker image that is run in an AWS Batch job that is dependent on a previous Batch job that runs the Sikraken container. It shares similarities to the Sikraken Dockerfile however the flex and bison packages are removed as they're unnecessary and instead requires Python to be installed to as the scripts to modify the .html files generated by existing Bash scripts made for visualising Sikraken's test run results are written in bash. Upon running the container, the script generate_reports.sh as seen in Appendix 12 is run.

```
FROM ubuntu:24.04

WORKDIR /app

RUN DEBIAN_FRONTEND=noninteractive apt-get update && apt-get install -y
\
    wget curl unzip gcc ca-certificates \
    python3 \
    bc \
    && rm -rf /var/lib/apt/lists/*

RUN curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
awscliv2.zip \
    && unzip awscliv2.zip \
    && ./aws/install \
    && rm -rf aws awscliv2.zip
```

```
COPY ReportScripts /app/ReportScripts
COPY SikrakenDevSpace/bin/helper /app/ReportScripts
COPY SikrakenPythonScripts /app/SikrakenPythonScripts

RUN chmod +x /app/ReportScripts/*
RUN chmod +x /app/SikrakenPythonScripts/*

ENTRYPOINT ["/app/ReportScripts/generate_reports.sh"]
```