Onboarding STeLLAR

Overview

STeLLAR is a benchmarking tool or framework for serverless system performance analysis. It takes 2 input files - a vendor configuration and an experiments configuration JSON file that can be customised to assess different providers under various testing conditions. These conditions can be decided/configured based on the component of the serverless infrastructure that is stressed to observe the effect of that on the median and tail latencies or response times.

The STeLLAR dashboard visualizes the results of three key types of experiments conducted on major providers, including AWS, Azure, Google Cloud Functions, and Cloudflare. These experiments are run continuously on a weekly basis through automated scheduled jobs and workflows, which are orchestrated using GitHub Actions and runners. Understanding GitHub Actions.

The workflows are configured to check out the `main` and `continuous-benchmarking` branches of the repository to package and deploy necessary functions and execute the experiments.

The workflows schedule is as follows:

Sunday, 23:00 UTC | Monday, 07:00 SGT: Provision VMs and set up self-hosted runners.
Monday, 00:00 UTC | Monday, 08:00 SGT: Continuous Benchmarking Baseline Experiments.
Monday, 12:00 UTC | Monday, 20:00 SGT: Continuous Benchmarking Image Size Experiments.
Tuesday, 00:00 UTC | Tuesday, 08:00 SGT: Continuous Benchmarking Runtime Experiments.
Wednesday, 00:00 UTC | Wednesday, 08:00 SGT: Teardown VMs and remove self-hosted runners.

Note: The workflow files are located under `.github/workflows` in the `main` branch. Baseline experiments are expected to complete on Monday (UTC), while Image Size and Runtime experiments should both conclude on Tuesday (UTC).
In the event of any job failures, a notification is automatically sent to a designated Slack channel. This notification includes information such as the workflow and job that has failed.

Understanding the Workflows

Provision VMs and setup self hosted runners: This workflow provisions VMs, installs the STeLLAR client, and sets up a github actions runner on the VM, for each provider. This is to ensure that the readings/results are unaffected by the propagation delays between STeLLAR client machine and its respective cloud provider. (For Cloudflare, since it does not provide VM services, we use another AWS EC2 instance to setup and run experiments directed to Cloudflare) (About self-hosted runners - GitHub Docs)

Continuous Benchmarking Experiments
Each workflow begins by building the STeLLAR client from the `main.go` binary executable, which the workflow retrieves from the `continuous-benchmarking` branch. This process packages and converts the `main.go` file into an executable, creating the STeLLAR tool.
During the experiments, the `./stellar` executable is run with specific flags tailored to the respective providers and the type of experiments being conducted.

Baseline invocations: Warm and Cold: These experiments assess the performance of serverless providers under two conditions:

1. **Warm Instances**: Requests are concentrated on a single or a few workers with a lower Inter-Arrival Time (IAT) between requests. This scenario evaluates the response times with instances that are already warm.
2. **Cold Instances**: Requests are distributed among many workers (e.g., 50) with a higher IAT, meaning most instances are likely free or cold. In this setting, requests are distributed among these cold instances, allowing us to measure and evaluate the response times with cold instances.

For example, the command below is used for warm function invocations on AWS Lambda:

```
cd src && ./stellar -a 356764711652 -o latency-samples -c
../continuous-benchmarking/experiments/warm-function-invocations/warm-baseline-aws
.json -db -w
```

The JSON file would specify the experiment configurations such as IAT in seconds and Parallelism to specify the number of concurrent workers in one sub experiment/one burst of requests. `latency-samples` is the file path where the output statistics should be written. The `db` flag is set when the experiment results need to be written to the dashboard to display.
To trace any issues at this step, refer to the `src/main.go` file in the `continuous-benchmarking` branch, then follow the flow through `src/setup/run.go`, `src/setup/serverless-config.go`, and `src/benchmarking/trigger.go`.

Image Size Experiments: For Image Size experiments, on the dashboard, we conduct and visualize results from evaluations with function image sizes of 50 MB and 100 MB for AWS, Azure, and Google Cloud. We configure the experiments config file with values 50 or 100 MB as function image size in MB and keep settings such that requests are sent to cold instances. This helps us assess the impact of image size on cold start latencies (time taken to load function image on cold instance)
For example, the command below is used for a 50 MB image size experiment on AWS Lambda:

```
cd src && ./stellar -a 356764711652 -o latency-samples -c
../continuous-benchmarking/experiments/cold-function-invocations/image-size/cold-i
mage-size-50-aws.json -db
```

Runtime Experiments: Runtime experiments are conducted to investigate the impact of the runtime language of deployed functions on cold start latencies. Functions are packaged along with their dependencies and deployed via Serverless to different providers. When these function images are loaded certain functions can potentially take longer to load depending on the language used at runtime.

`cd src && ./stellar -o latency-samples-aws -l debug -c
../continuous-benchmarking/experiments/cold-function-invocations/language-runtime-deployment-method/aws/
cold-hello${{ matrix.runtime }}-zip-aws.json -db`

Teardown VMs and remove self-hosted runners: Finally at the end of all the experiments, this workflow deletes the VM, removes the self hosted runner.

Issues to Address:

AWS Runtime Experiment (Java) Failure -

With Java runtime, during deployment the CPU usage hits 99% and the operation gets cancelled. Due to this the runner on the EC2 instance is affected and goes offline becoming unable to take up any queued workflows.

This time, when I tried restarting the runner in a new instance and re ran the workflows, Java runtime kept falling at the deployment step, while other workflows ran without errors. Earlier failures were not specific to the Java runtime experiment.

Finally on re running the workflow after 2-3 hours after failure (the runner was offline, but the instance gained a few CPU credits) the deployment step executed successfully with only around 20% CPU usage. So, not sure why this is not consistent and if the credits accumulation has a role to play.

We can try a different option with more RAM, and see if the issue persists.

Function Deployment Methods for AWS and Azure -

Currently both use Serverless Framework V3, however, support for this version will be discontinued by the end of this year, and the a) new versions will no longer support Azure plugins (any non AWS services)
For more details, refer to Upgrading to Serverless Framework V4

AWS and Azure use the `sls deploy` command for deployment. This process involves:

1. Using the experiment configuration file to generate a `serverless.yml` file.
2. Running the `sls deploy` command within the directory containing the `serverless.yml` file, which returns the list of function URLs with HTTP endpoints for sending requests.

Compare the Google Cloud deployment code found in `src/setup/serverless-config.go` with the deployment method for Azure. The Azure deployment process follows the `DeployService` function.

```go
// DeployService deploys the functions defined in the serverless.com file
func DeployService(path string) string {
	log.Infof(fmt.Sprintf("Deploying service at %s", path))
	slsDeployCmd := exec.Command("bash", "-c", "sls deploy")
	slsDeployCmd.Dir = path
	slsDeployMessage := util.RunCommandAndLogWithRetries(slsDeployCmd, 3)
	return slsDeployMessage
}

// DeployGCRContainerService deploys a container service to cloud provider
func (s *Serverless) DeployGCRContainerService(subex *SubExperiment, index int, randomTag string, imageLink string, path string, region string) {
	log.Infof("Deploying container service(s) to GCR...")
	for i := 0; i < subex.Parallelism; i++ {
		name := fmt.Sprintf("%s-%s", randomTag, createName(subex, index, i))
		providerFunctionNames["gcr"] = append(providerFunctionNames["gcr"], name) // Used for function removal

		gcrDeployCommand := exec.Command("gcloud", "run", "deploy", name, "--image", imageLink, "--allow-unauthenticated", "--region", region)
		deployMessage := util.RunCommandAndLog(gcrDeployCommand)
		subex.Endpoints = append(subex.Endpoints, EndpointInfo{ID: GetGCREndpointID(deployMessage)})
		subex.AddRoute("")
	}
}
```

In src/setup/serverless-config.go in branch, continuous-benchmarking

Alternatives for Azure: We need to explore a similar deployment (like GCR) approach for Azure, potentially using the CLI or another method, to replace the deprecated Serverless Framework V3.

You can work on this in a new branch. In this branch, once you change this and want to test, you can create a new workflow (specifically with an Azure job/experiment) or modify an existing workflow by commenting out non-Azure experiments/jobs to focus solely on Azure and run it to see if it works, any errors, etc.

Authentication via Access Keys for AWS: Serverless V4 will require authentication via the Serverless Framework Dashboard or a Serverless License Key, impacting AWS workflows. This change mandates that all users, both free and paid, authenticate through the CLI.
"To use this method in CI/CD pipelines, go to the Access Keys view in Serverless Framework Dashboard, create a new Access Key for CI/CD use, and save it as the `SERVERLESS_ACCESS_KEY` environment variable in your CI/CD pipeline."
This `SERVERLESS_ACCESS_KEY` can be used as an Action's Secret in the workflow for AWS experiments.