# Globally distributed load tests in Azure with Locust

Heyko Oelrichs
Mar 30 · 7 min read

I recently came across the challenge to conduct massive distributed load tests on an scalable application hosted in Azure. After doing some research on the web I found a couple of viable tools, including Locust which I had already heard of in one of my recent customer projects. I therefore digged a bit deeper into Locust to learn more about it and its capabilities and here are my results.

> *Locust is an easy to use, scriptable and scalable open source load and performance testing tool. You define the behaviour of your users in regular Python code, instead of using a clunky UI or domain specific language. This makes Locust infinitely expandable and very developer friendly.*

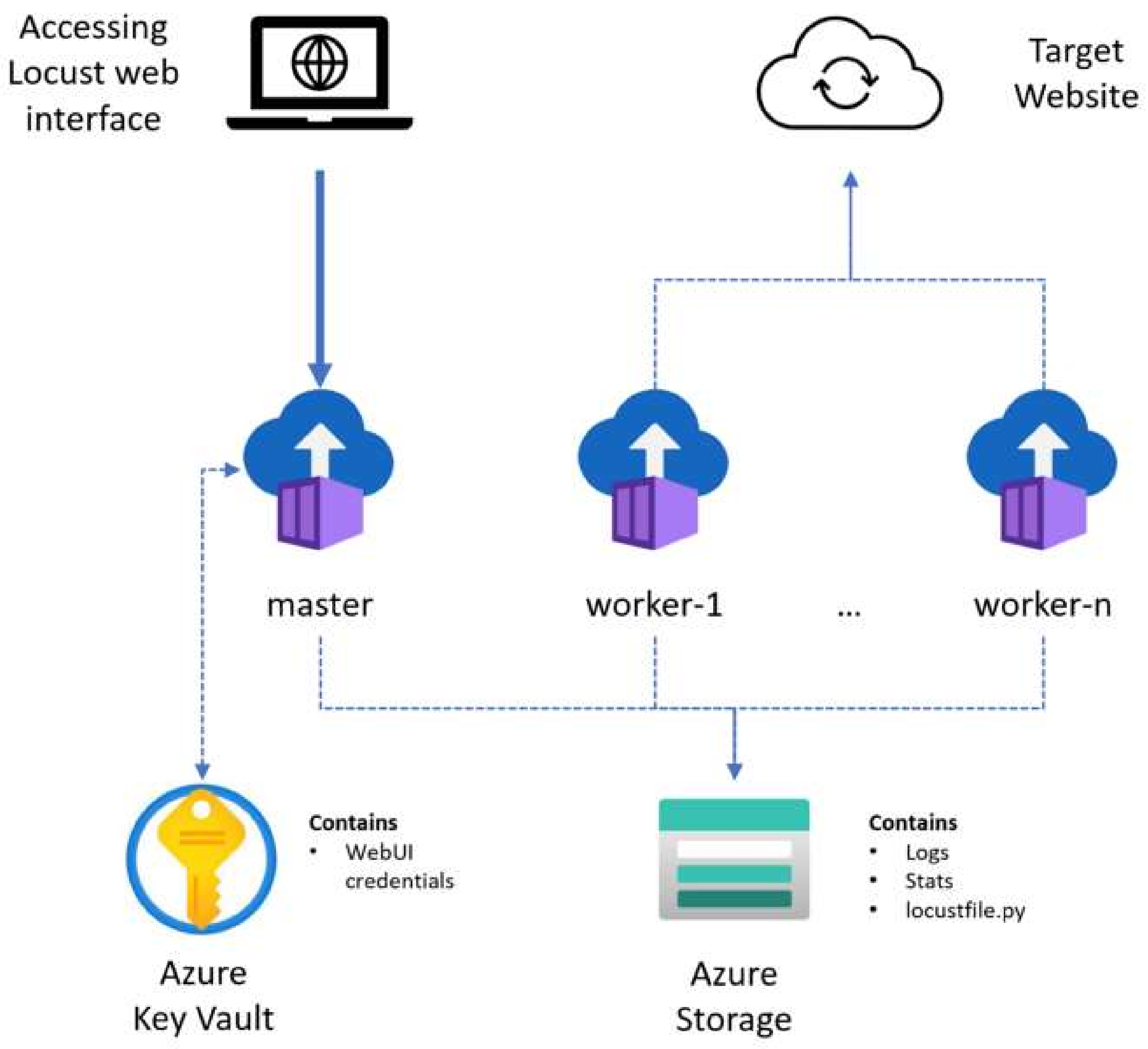
One of the first blog posts I found, related to Locust on Azure, was Davide Mauri's "Running Locust on Azure" here on medium.com. He describes pretty well how to run Locust on Azure Container Instances (ACI) using Azure Resource Manager (ARM) Templates to deploy the required components and gave me a good starting point. I, other than Davide, decided to use Terraform instead of ARM to deploy my infrastructure as this fit better into my existing setup and added a couple improvements you will see further down.

Locust consists of master and worker nodes, the master node is doing the test orchestration and is hosting a web interface (it can also be used in a headless mode for example as part of a CI/CD pipeline, more about that further down) where we can start tests and see live statistics, the worker nodes are running the tests itself, simulating our users. Locust can be used on a single machine as well as distributed using Docker containers. We are going for the container option and we therefore need a solution to host our containers, in my case I have chosen Azure Container Instances (ACI).

The reason why I've decided to use ACI instead of Azure Kubernetes Service (AKS) or other platforms is the fact that ACI is a Platform-as-a-Service (PaaS) offering in Microsoft Azure and "serverless", that means that it comes completely without any infrastructure we have to set up and maintain and therefore without any overhead, we can spin up our master and worker nodes on-demand and scale them down to zero when we do not need them. This gives us full flexibility and also saves money.

In addition to that we need a place to store our locustfile.py which contains our test definition. I'm using an Azure Storage Account file share for that, all our container instances will mount it to load the test definition from there. I've also added an Azure Key Vault to store the credentials used for the Locust web interface authentication in a secure way instead of writing them in plain text into my Terraform definition.

The overall architecture in the end will look like this:



Locust on ACI — Architecture Overview

Before we dive deeper into the deployment itself, why does it make sense to run tests distributed, using multiple instances in different geographical regions?

- First of all do we want to have a realistic usage pattern, typical applications, especially the ones with global coverage and usage, are accessed by users in various locations in different geographical regions. This is even more important when the application itself is deployed across multiple regions using global load balancing technologies and services like Azure Traffic Manager, Azure Front Door, AWS Route 53 or others.
- The other, more technical reason are the technical limitations of a single instance. A single Virtual Machine or a single Container is limited in multiple ways, it has only a fix number of ports, it is limited by its CPU and memory as well as bandwidth. Other limitations might be network latency, throughput and the fact that all requests are coming from a single IP or IP range.

Using a larger number of instances, spread across multiple data centers and geographical regions adresses the various limitations listed above, helping us to have a more realistic usage pattern for our load tests.

## Deploy Locust on Azure

Let us now take a look into how to deploy Locust. Starting with the master node, our Terraform definition (below) will create a single master instance (if ***var.workers*** is ≥ 1). It also allows us to scale to zero by setting ***var.workers*** to 0. It is using a container instance with a public DNS name, accessible on port 8089 (which is the default port for the locust web interface) and is protected using basic web-auth with its password stored in Azure Key Vault (more details further down below).

```
resource "azurerm_container_group" "master" {
  count               = var.locustWorkerNodes >= 1 ? 1 : 0
  name                = "${random_pet.deployment.id}-locust-master"
  location            = azurerm_resource_group.deployment.location
  resource_group_name = azurerm_resource_group.deployment.name
  ip_address_type     = "Public"
  dns_name_label      = "${random_pet.deployment.id}-locust-master"
  os_type             = "Linux"
  os_type              = "Linux"

  container {
    name   = "${random_pet.deployment.id}-locust-master"
    image  = var.locustVersion
    cpu    = "2"
    memory = "2"

    commands = [
        "locust",
        "--locustfile",
        "/home/locust/locust/${azurerm_storage_share_file.locustfile.name}",
        "--master",
        "--web-auth",
        "locust:${azurerm_key_vault_secret.locustsecret.value}",
        "--host",
        var.targeturl
    ]

    volume {
        name = "locust"
        mount_path = "/home/locust/locust"

        storage_account_key  = azurerm_storage_account.deployment.primary_access_key
        storage_account_name = azurerm_storage_account.deployment.name
        share_name           = azurerm_storage_share.locust.name
    }

    ports {
      port     = "8089"
      protocol = "TCP"
    }

    ports {
      port     = "5557"
      protocol = "TCP"
    }

  }

  tags     = local.default_tags
}
```

locust-master-terraform.tf hosted with ♡ by GitHub view raw

Our worker nodes are defined in a pretty similar way with some small differences, they can scale between 0 and n (only limited by the available container groups in your subscription), they do not have a web interface and they communicate with the master node on port 5557/TCP. Our workers also do not have a public DNS name.

The port definition below is not really needed, the reason it is in here is that TF requires it.

```
resource "azurerm_container_group" "worker" {
  count               = var.locustWorkerNodes
  name                = "${random_pet.deployment.id}-locust-worker-${count.index}"
  location            = var.locustWorkerLocations[count.index % length(var.locustWorkerLocatior
  resource_group_name = azurerm_resource_group.deployment.name
  ip_address_type     = "Public"
  os_type             = "Linux"

  container {
    name   = "${random_pet.deployment.id}-worker-${count.index}"
    image  = var.locustVersion
    cpu    = "2"
    memory = "2"

    commands = [
        "locust",
        "--locustfile",
        "/home/locust/locust/${azurerm_storage_share_file.locustfile.name}",
        "--worker",
        "--master-host",
        azurerm_container_group.master[0].fqdn
    ]

    volume {
        name = "locust"
        mount_path = "/home/locust/locust"

        storage_account_key  = azurerm_storage_account.deployment.primary_access_key
        storage_account_name = azurerm_storage_account.deployment.name
        share_name           = azurerm_storage_share.locust.name
    }

    ports {
      port     = 8089
      protocol = "TCP"
    }

  }

  tags     = local.default_tags
}
```

locust-worker-terraform.tf hosted with ♡ by GitHub view raw

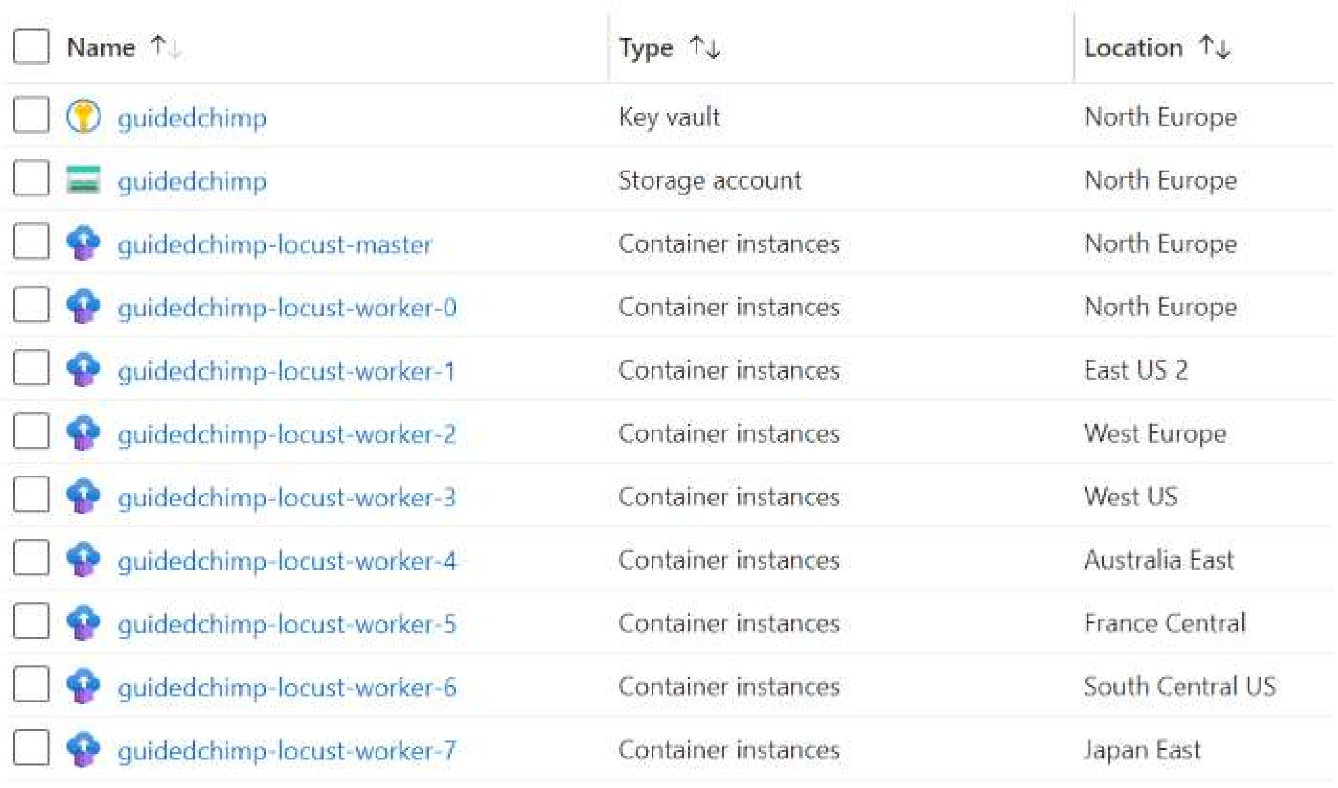
Interesting to note in the definition above, in line 4 (in locust-worker-terraform.tf), is the use of a list of worker_locations. This list contains all the Azure Regions we want to use for our load tests (see locust-variables-terraform.tf below). Our workers will be spread across those regions randomly based on the number of workers we are going to deploy.

```
variable "location" {
  description = "The Azure Region in which the master and the shared storage account will be pr
  type        = string
  default     = "northeurope"
}

variable "environment" {
  description = "Environment Resource Tag"
  type        = string
  default     = "dev"
}

variable "targeturl" {
  description = "Target URL"
  type        = string
  default     = "https://my-sample-application.com"
}

variable "locustVersion" {
  description = "Locust Container Image Version"
  type        = string
  default     = "locustio/locust:1.4.3"
}

variable "locustWorkerNodes" {
  description = "Number of Locust worker instances (zero will stop master)"
  type        = string
  default     = "0"
}

variable "locustWorkerLocations" {
  description = "List of regions to deploy workers to in round robin fashion"
  type        = list
  default     = ["northeurope", "eastus2", "westeurope", "westus", "australiaeast", "francecent
}

```

```
variable "prefix" {
  description = "A prefix used for all resources in this example. Must not contain any special
  type        = string
  validation {
    condition     = length(var.prefix) >= 5 && length(var.prefix) <= 10
    error_message = "Prefix must be between 5 and 10 characters long."
  }
}
```

locust-variables-terraform.tf hosted with ♡ by GitHub view raw

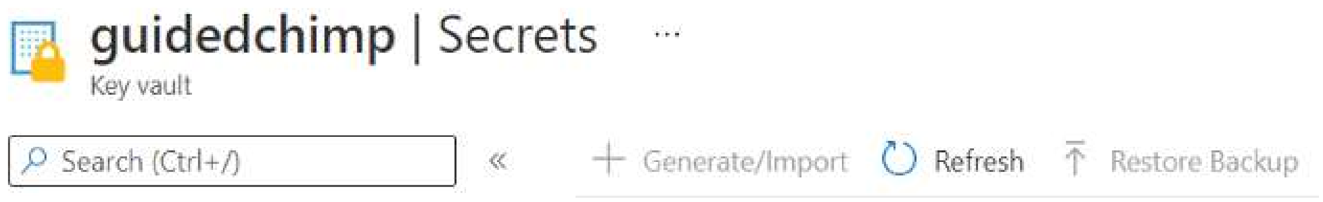
The end result of our deployment looks like this, with eight worker nodes, one master node as well as an Azure Storage Account and Azure Key Vault:



| Name | Type | Location |
|---|---|---|
| guidedchimp | Key vault | North Europe |
| guidedchimp | Storage account | North Europe |
| guidedchimp-locust-master | Container instances | North Europe |
| guidedchimp-locust-worker-0 | Container instances | North Europe |
| guidedchimp-locust-worker-1 | Container instances | East US 2 |
| guidedchimp-locust-worker-2 | Container instances | West Europe |
| guidedchimp-locust-worker-3 | Container instances | West US |
| guidedchimp-locust-worker-4 | Container instances | Australia East |
| guidedchimp-locust-worker-5 | Container instances | France Central |
| guidedchimp-locust-worker-6 | Container instances | South Central US |
| guidedchimp-locust-worker-7 | Container instances | Japan East |

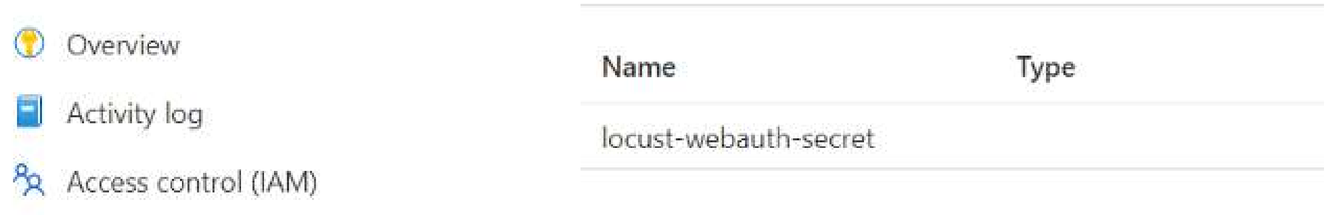
Locust ACI deployment with 8 worker nodes

Our Locust web interface is accessible on the locust-master container instance FQDN on port 8089 (Terraform will also print out its full FQDN in its outputs). It is protected using basic web-auth, the username is set to “locust” and the password is stored in Azure KeyVault:



guidedchimp | Secrets ···

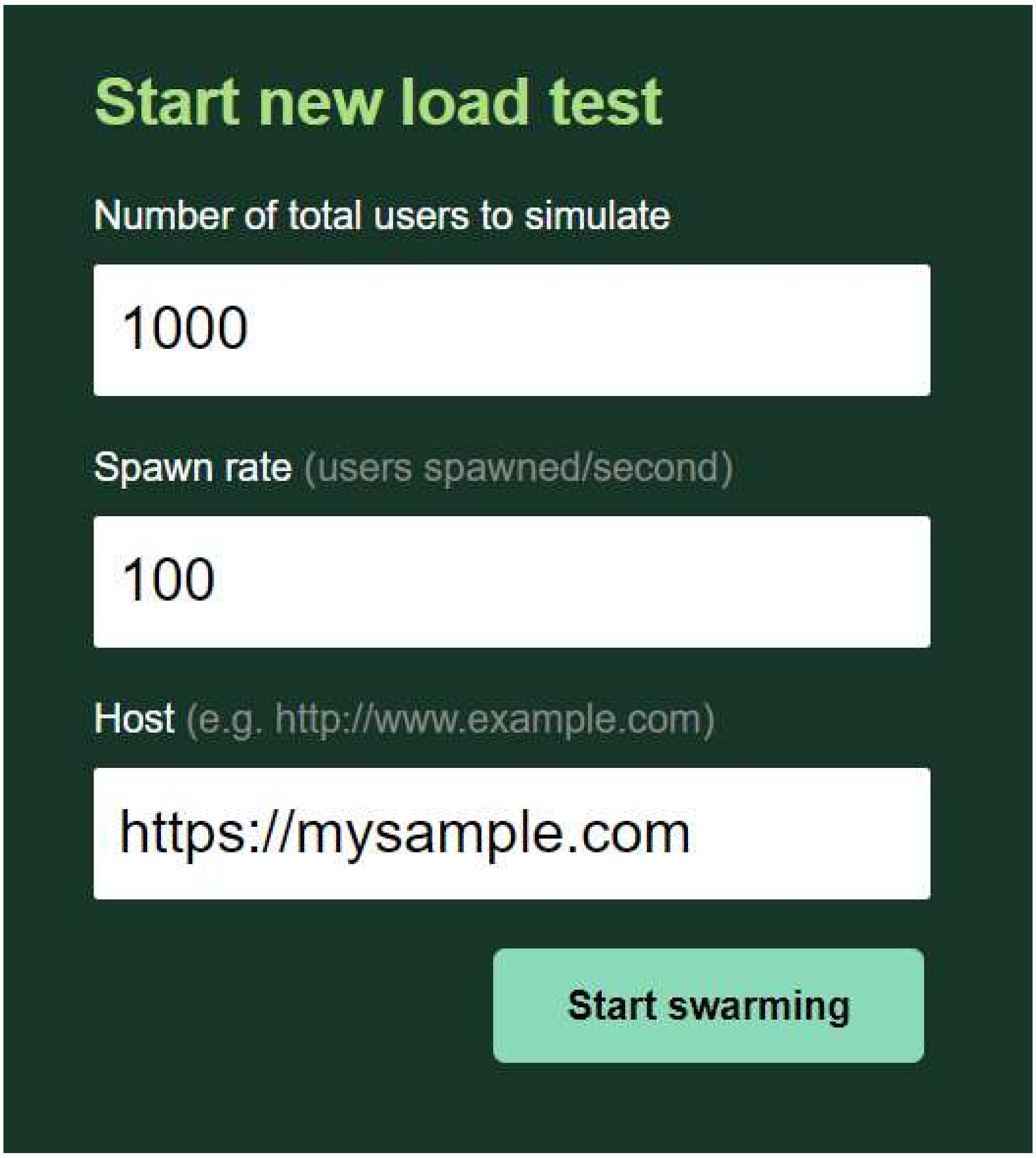
Key vault



| Name | Type |
|---|---|
| locust-webauth-secret | |

Locust web-auth password stored in Azure Key Vault

Accessing the Locust web-interface will bring up the following dialog that lets you configure the total number of users to simulate, the spawn rate as well as the target host:


Start new load test

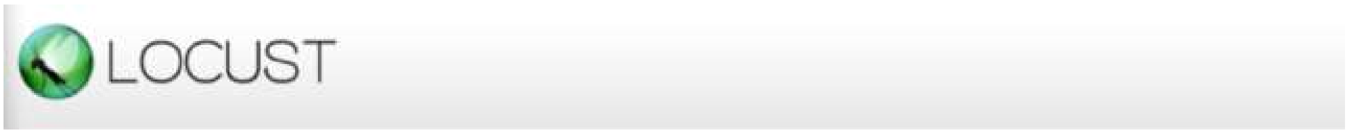
Number of total users to simulate: 1000

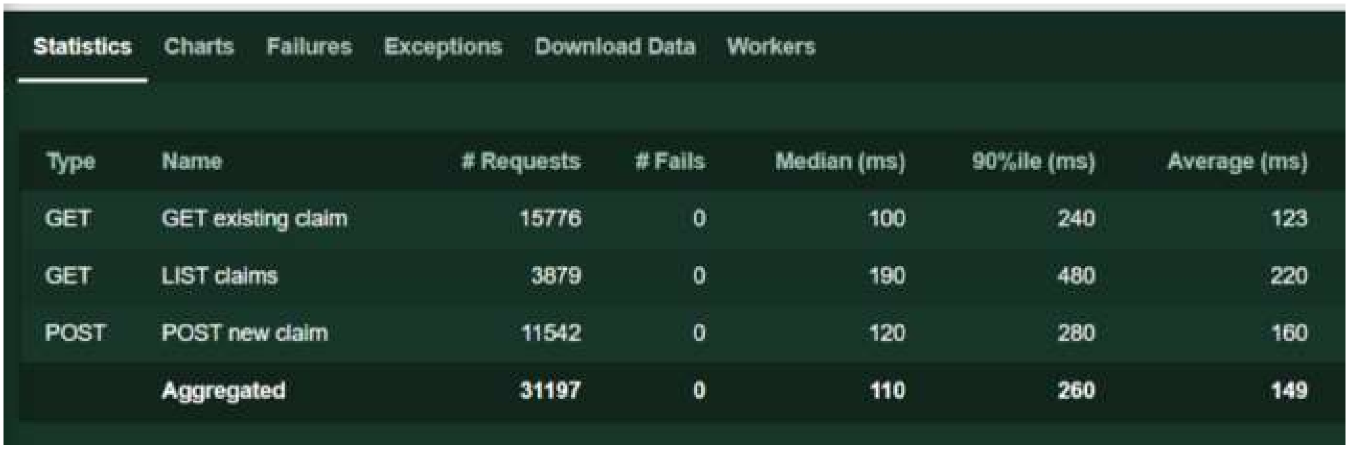
Spawn rate (users spawned/second): 100

Host (e.g. http://www.example.com): https://mysample.com

Locust “Start new load test” dialog

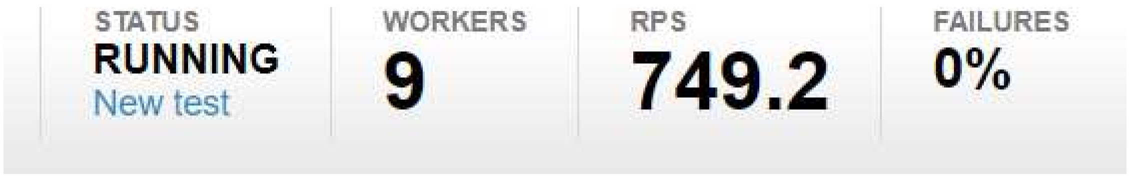
After starting a new test we are seeing our load test results coming in in near realtime:




Statistics
Charts
Failures
Exceptions
Download Data
Workers

| Type | Name | # Requests | # Fails | Median (ms) | 90%ile (ms) | Average (ms) |
|---|---|---|---|---|---|---|
| GET | GET existing claim | 15776 | 0 | 100 | 240 | 123 |
| GET | LIST claims | 3879 | 0 | 190 | 480 | 220 |
| POST | POST new claim | 11542 | 0 | 120 | 280 | 160 |
| | Aggregated | 31197 | 0 | 110 | 260 | 149 |

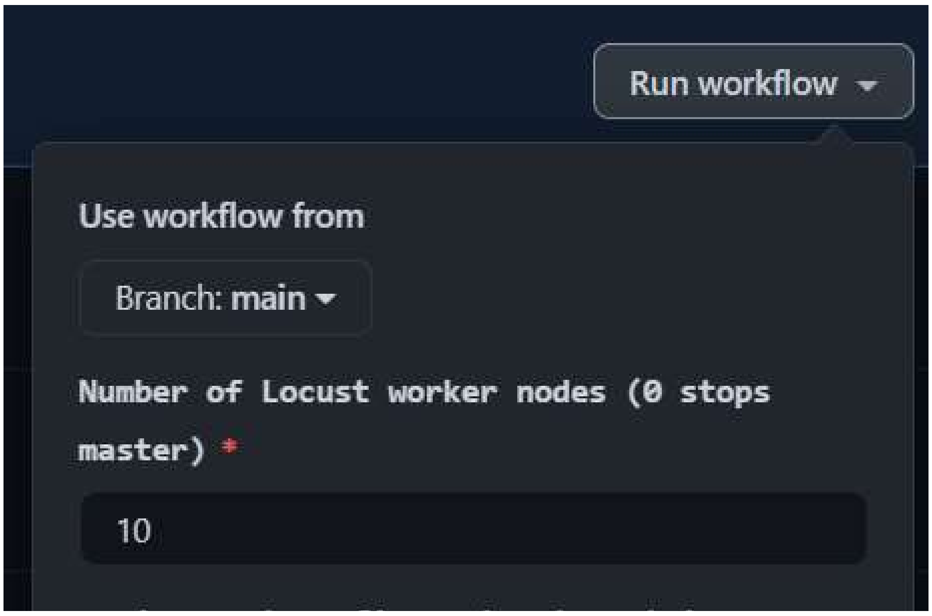
As well as details about our setup, including the number of worker nodes, requests per second and failure rate:



| STATUS | WORKERS | RPS | FAILURES |
|---|---|---|---|
| RUNNING New test | 9 | 749.2 | 0% |

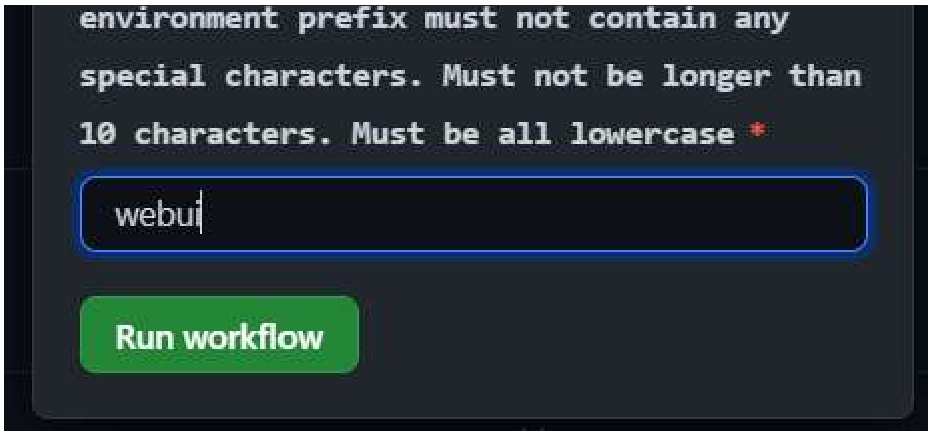
## How to further automate Locust?

Now that we have seen how to deploy Locust via Terraform on ACI to conduct tests via the web interface, how can we further automate the use of Locust? One option you can see in my repository is to deploy Locust via GitHub Actions, you can scale Locust up when needed, conduct load tests via the web interface and scale it down afterwards. That’s a viable way to spin up your test infrastructure on demand.



Number of Locust worker nodes (0 stops master) *

10

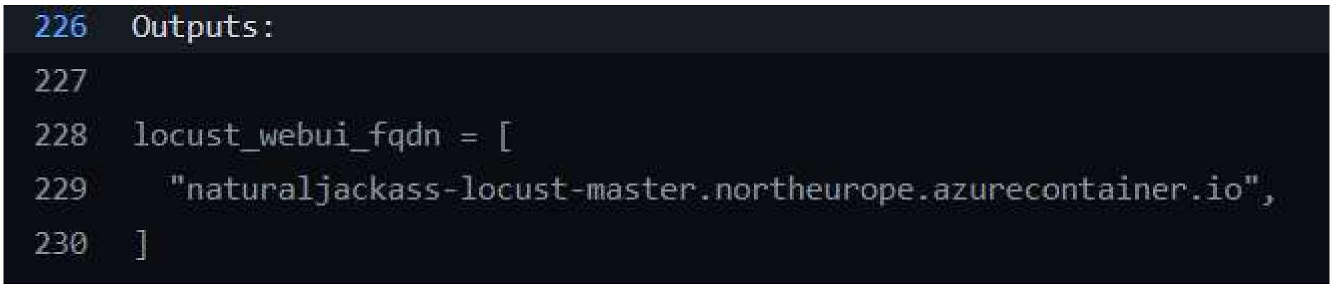


environment prefix must not contain any special characters. Must not be longer than 10 characters. Must be all lowercase *

webui

Deploy Locust via a GitHub workflow

The master's public DNS name will be shared at the end of a successful workflow run as an output of Terraform, the password to access the web interface is stored in Azure Key Vault, as described above:

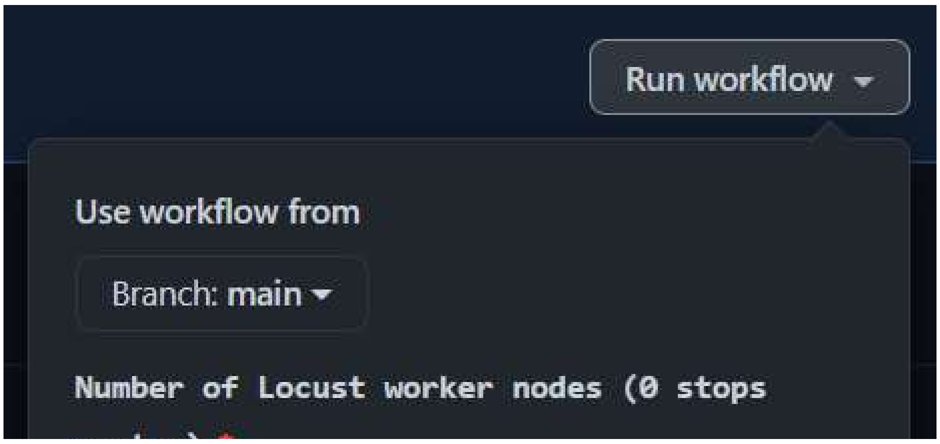


```
Outputs:

locust_webui_fqdn = [
  "naturaljackass-locust-master.northeurope.azurecontainer.io",
]
```

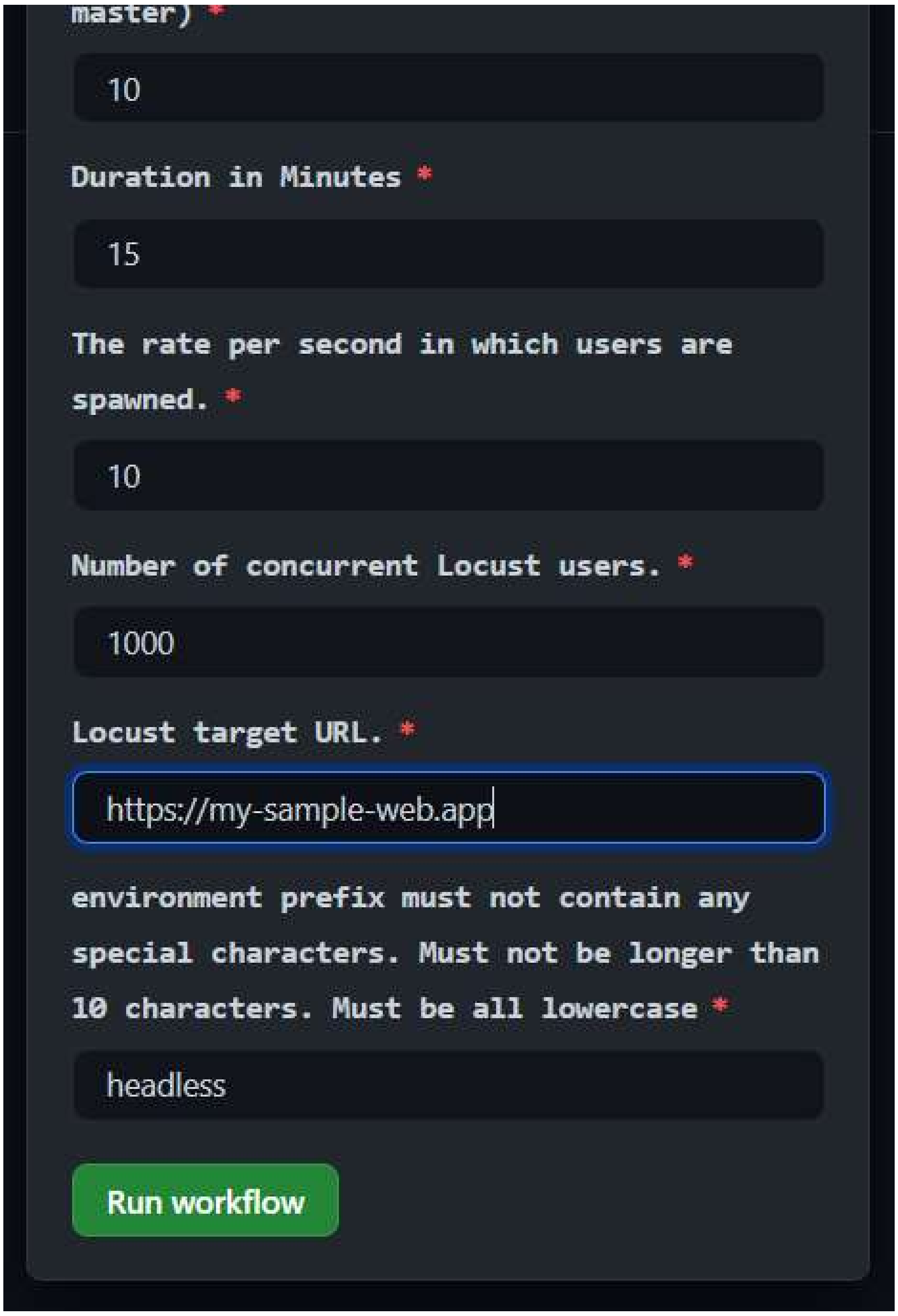
Locust master public DNS name

You can use the same GitHub workflow to scale down the environment by running the same pipeline again, setting the "Number of Locust worker nodes" to 0 which will remove all infrastructure except the Storage Account and Key Vault.

But what about even more automation? As mentioned above, Locust does also support a so called "headless" mode. In headless mode we do not have a web interface and specify more things like the number of nodes, spawn time, run time etc. upfront, before we deploy our test infrastructure.



Number of Locust worker nodes (0 stops


master) *

10

Duration in Minutes *

15

The rate per second in which users are spawned. *

10

Number of concurrent Locust users. *

1000

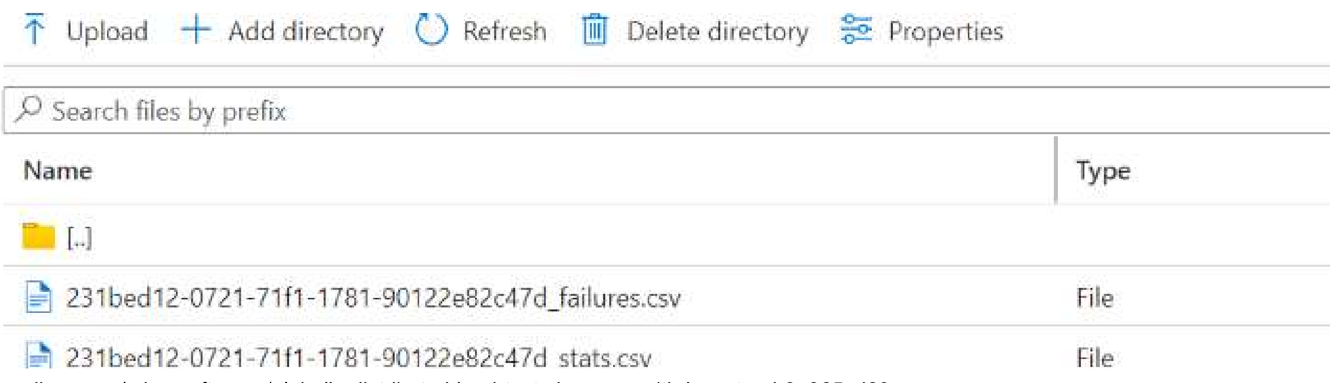
Locust target URL. *

https://my-sample-web.app

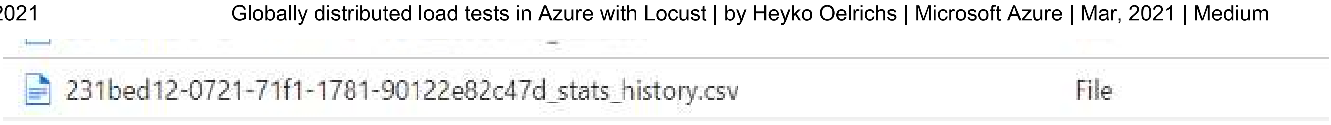
environment prefix must not contain any special characters. Must not be longer than 10 characters. Must be all lowercase *

headless

This workflow will spin up the required infrastructure in Azure, conduct the load test as defined when starting the workflow, write the test results into its Storage Account and will tear down the infrastructure (except of the Storage Account that contains the test results).

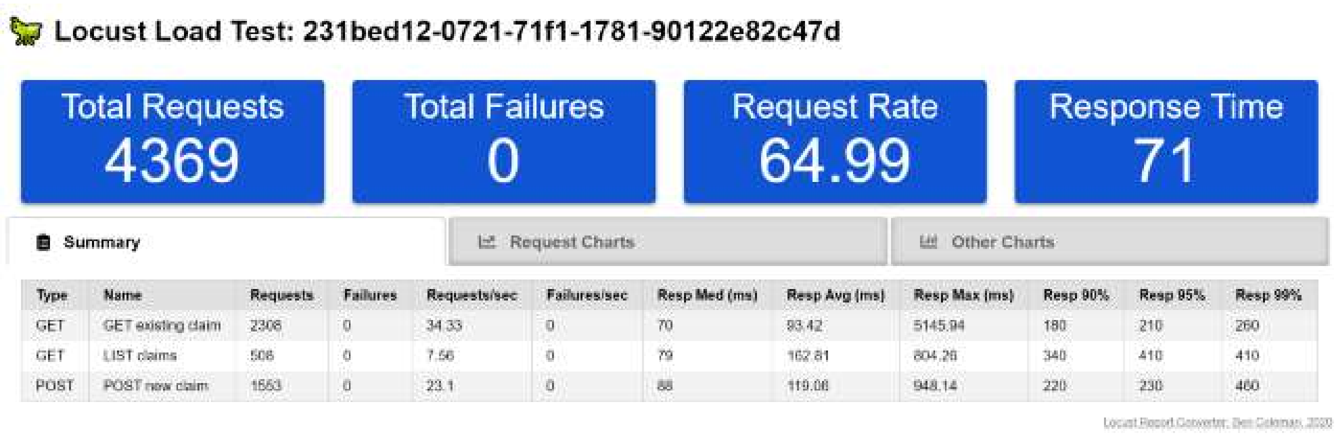


| Name | Type |
|---|---|
| [..] | |
| 231bed12-0721-71f1-1781-90122e82c47d_failures.csv | File |
| 231bed12-0721-71f1-1781-90122e82c47d_stats.csv | File |


231bed12-0721-71f1-1781-90122e82c47d_stats_history.csv File

Locust stats stored in Azure Storage

As you can see in the previous screenshot are our tests results written to Azure Storage as CSV files — in case you want to convert these test results into an easier to read HTML report I recommend to take a look on to the Locust HTML Report Converter. It's a simple Go command line application to convert the results into a HTML report.


Locust Load Test: 231bed12-0721-71f1-1781-90122e82c47d

| Total Requests | Total Failures | Request Rate | Response Time |
|---|---|---|---|
| 4369 | 0 | 64.99 | 71 |

| Type | Name | Requests | Failures | Requests/sec | Failures/sec | Resp Med (ms) | Resp Avg (ms) | Resp Max (ms) | Resp 90% | Resp 95% | Resp 99% |
|---|---|---|---|---|---|---|---|---|---|---|---|
| GET | GET existing claim | 2308 | 0 | 34.33 | 0 | 70 | 93.42 | 5145.94 | 180 | 210 | 260 |
| GET | LIST claims | 508 | 0 | 7.56 | 0 | 79 | 162.81 | 804.26 | 340 | 410 | 410 |
| POST | POST new claim | 1553 | 0 | 23.1 | 0 | 88 | 119.08 | 948.14 | 220 | 230 | 460 |

Locust Report Converter: Ben Coleman, 2020

Locust HTML Report Converter — Example Report

If you want to see more of my Terraform code including the code for a headless Locust deployment, the discussed GitHub workflows etc. please visit my locust-on-aci repository on GitHub. You'll find all the components you need to setup the exact same deployment that I've described here.

In case you're wondering where the fancy names like 'guidedchimp' and others in my previous screenshots came from, for deployments that require flexibility and to avoid naming conflicts i usually use the random_pet resource of the Terraform random provider.

Azure Load Testing Microsoft Cloud Github