Project 3: Serverless Computing and Monitoring Dashboard

Introduction

A serverless function is a function written by a software developer for a single purpose. It is a programmatic feature of serverless computing, where the general idea is to build a piece of business logic that is both stateless (does not maintain data) and ephemeral (is used and destroyed as needed). Serverless functions are hosted and managed on infrastructure provided by cloud computing companies. These companies take care of code maintenance and execution so that developers can deploy new code faster.

To use serverless functions, all a developer has to do is write the function code and deploy it to a managed environment. A typical serverless function process is exemplified in Figure 1.

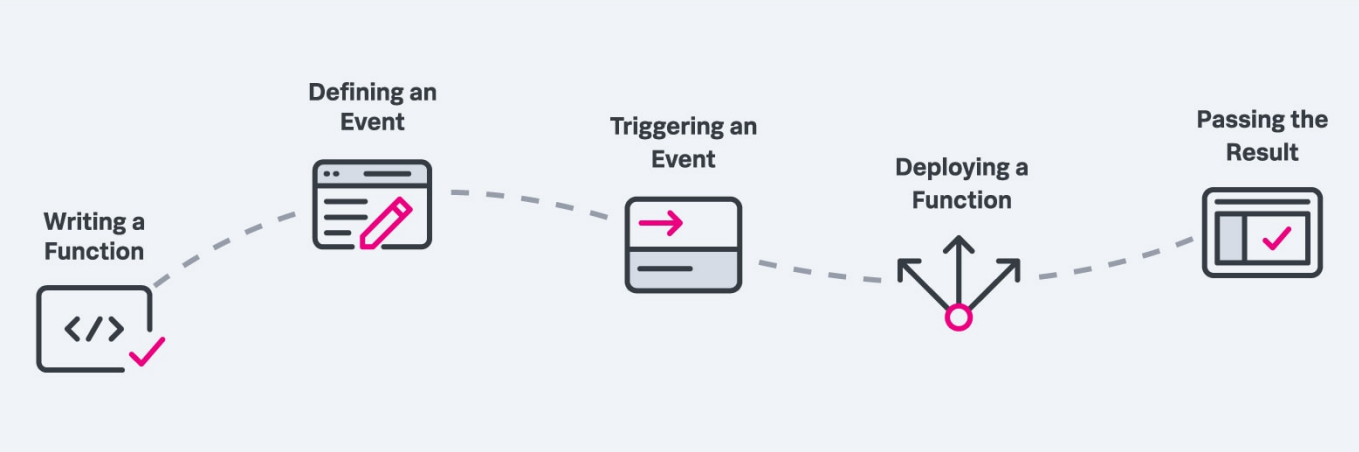

Figure 1. Standard serverless function process. Source: Slunk.

The developer writes a function that fulfills a specific purpose in the application code, such as a form mailer or measurement analysis, and defines an event that will trigger the cloud-native service provider to execute the function (for instance, an HTTP request or a time interval). When an event is triggered, the cloud service provider starts a new instance of the function if none are running. Finally, the result of the executed function is sent to the user, another function, or persisted in storage.

This paradigm is supported in many cloud providers like Google Cloud Functions (Google), AWS Lambda (Amazon AWS), and Azure Functions (Microsoft Azure), and has been used in diverse use cases such as Web applications, data manipulation, stream processing at scale, and continuous integration pipelines.

General Goal

In this practical project, students will implement and deploy a serverless function to analyze resource usage of the course's virtual machine. Students will then implement a dashboard to continuously display resource use or create a simplified runtime for executing functions.

Overview

We have made available a process for collecting the VM's usage statistics to be used as your stream input data. Each new record is collected periodically every 5 seconds and stored/replaced in the Redis key-value

storage available in the VM. The data is stored in Redis under the `metrics` key.

> Redis is an open source, in-memory key-value store, used as a database, cache, and message broker. Redis provides data types such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, geospatial indexes, and streams. Redis is commonly used as an alternative to Apache Kafka when speed is more important, but the amount of data that is going to be dealt with is not large.

In Task 1, students will create a serverless function to process the resource usage metrics monitored on the VM. The results from the serverless function will be stored on Redis. In Task 2, students will implement and deploy a monitoring dashboard to display the monitored information computed in Task 1. Finally, in Task 3, students will implement a simplified runtime to support serverless functions.

> Task 2 will provide experience with one of the many existing monitoring frameworks and how to get them deployed on the cloud, which may be useful for data scientists. Task 3 will provide experience with the infrastructure used to support serverless functions in the cloud, which may be useful to back-end software engineers.

Group work

This assignment may be submitted individually or in groups of up to two students.

Task 1: Serveless Function and Runtime

Students will create and use a serveless function to process the periodic resource use measurements.

Data Input

The system collects resource usage information using the `psutil` module. Measurements are stored in a key named `metrics` under Redis, but are read by the serverless framework and passed on to your serverless function as a dictionary. More precisely, each call to your serverless function will receive as parameter a resource usage measurement in a JSON object containing the following keys:

- `timestamp`: A string containing the time of the current measurement.
- `cpu_freq_current`: A float representing the current CPU frequency (in MHz).
- `cpu_percent-X`: A float representing the utilization of CPU X (in %).
- `cpu_stats-ctx_switches`: The number of context switches (voluntary + involuntary) since boot.
- `cpu_stats-interrupts`: The number of interrupts since boot.
- `cpu_stats-soft_interrupts`: The number of software interrupts since boot.
- `cpu_stats-syscalls`: The number of system calls since boot.
- `n_pids`: Number of the running PIDs.
- `virtual_memory-total`: The total physical memory.
- `virtual_memory-available`: Amount of memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory values depending on the platform and it is supposed to be used to monitor actual memory usage in a cross platform fashion.
- `virtual_memory-percent`: A float representing the memory utilization as a percentage.
- `virtual_memory-used`: Memory used, calculated differently depending on the platform and intended for informational purposes only.
- `virtual_memory-free`: Memory not being used at all that is readily available.
- `virtual_memory-active`: Memory currently in use or very recently used.

- `virtual_memory-inactive`: Memory that is marked as not used.
- `virtual_memory-buffers`: Cache for things like I/O buffers.
- `virtual_memory-cached`: Cache for things like filesystem inodes and data read from disk.
- `virtual_memory-shared`: Memory that may be simultaneously accessed by multiple processes.
- `virtual_memory-slab`: In-kernel data structures cache.
- `net_io_counters_eth0-bytes_sent1`: Number of bytes sent.
- `net_io_counters_eth0-bytes_recv1`: Number of bytes received.
- `net_io_counters_eth0-packets_sent1`: Number of packets sent.
- `net_io_counters_eth0-packets_recv1`: Number of packets received.
- `net_io_counters_eth0-errin1`: Total number of errors while receiving.
- `net_io_counters_eth0-errout1`: Total number of errors while sending.
- `net_io_counters_eth0-dropin1`: Total number of incoming packets which were dropped.
- `net_io_counters_eth0-dropout1`: Total number of outgoing packets which were dropped.

Check out the `psutil` documentation for further information.

Data Output (Computed Metric and Function Results)

Your serverless function should compute at least the average utilization of each CPU over the last hour and over the last minute. In other words, you should implement 2 moving averages to estimate the average utilization of each CPU, for a total of 2P metrics, where P is the number of CPUs. In addition to the average utilization, you should also define and implement another metric of your choice.

Your function should return a JSON-encodable dictionary containing at least 2P + 1 keys. A dictionary can be encoded in JSON if its keys are strings and its values are of type `None`, `int`, `float`, `str`, `list`, and `dict`, where the contained `list`s and `dict`s are also JSON encodable. The keys should describe the output metric (for example `avg-util-cpu1-60sec` or `avg-network-mbps-out-60sec`), and the values should be JSON-encodable objects that can be consumed by a monitoring dashboard (see Task 2).

Context Information

Your serverless function will also receive context information in addition to the input data described above. More precisily, the `context` object contains the following fields:

- `host`: Hostname of the server running Redis.
- `port`: Port where the Redis server is listening.
- `input_key`: Input key used to read monitoring data from Redis.
- `output_key`: Output key used to store metrics on Redis.
- `function_getmtime`: Timestamp of the last update to your module's Python file.
- `last_execution`: Timestamp of last execution of your serverless function and result storage on Redis.
- `env`: A JSON-encodable dictionary that persists between calls to the your serverless function. This variable can be used to persist small amounts of user data (context environment) between executions.

All fields except `env` are read-only and provided for information only. Changes made to the `env` dictionary will persist between calls to your function.

> When computing a moving average, you can store the last value of each moving average in `env` so you can update the moving average on subsequent calls to your function.

Integration with the Serverless Framework

The interface we use in our runtime is inspired by that of Amazon AWS Lambda. The implementation of the serverless function must follow specific requirements set by the runtime used in this project:

1. The serverless function must be at the root of a Python module. The module must be `import`able in Python.

2. The serverless function must be called `handler` and have the following definition:

```
def handler(input: dict, context: object) -> dict[str, Any]
```

The `input` parameter captures the monitoring data read from Redis and parsed into a dictionary (with the fields described above). The `context` parameter is a Python `object` that can be used to retrieve metadata about the deployment and possibly persist some small user-defined information (also described above). The `return`ed value should be a JSON-encodable dictionary containing the data to be plotted in a dashboard.

As in serverless computing platforms, the runtime will take care of receiving new resource usage measurements from Redis, calling our function, and storing the `return`ed results back in Redis. Your only task is to process the data in the serverless function. The result `return`ed by your function will be automatically persisted in Redis by the serverless framework.

A Docker image named `lucasmsp/serverless:redis` is provided and implements the runtime for the serverless function. It will load your module and call the `handler()` function for each resource usage measurement. We also provide a deployment file that you should use to deploy your application. You must use the image and the deployment files as provided.

Your tasks are to implement the module with the `handler()` function and create two Kubernetes ConfigMaps to integrate your module with the runtime image and deployment file. In particular, you need to create two ConfigMaps:

1. Create a ConfigMap named `pyfile` in your Kubernetes namespace. This ConfigMap should have a single key named `pyfile`, and the value should be your module's source code.

> A file can be mounted inside a Kubernetes Pod via a ConfigMap. A ConfigMap is a piece of information that maps a key to a value. We can use a ConfigMap in a Deployment to create a file containing the contents of a ConfigMap value (see the provided `deployment.yaml` file for an example). This is the mechanism used by the runtime to read your Python module containing the `handler` function.
>
> You must create a ConfigMap named `pyfile` in your Kubernetes namespace. This ConfigMap should have a single key named `pyfile`, and the value should be your module's source code. You can generate the ConfigMap using the following command:
>
> ```
> # format: kubectl create configmap <name> --from-file <key>=
> <file-path> --output yaml
> kubectl create configmap pyfile --from-file pyfile=mymodule.py --
> output yaml
> ```

> After you generate or update the ConfigMap, you can use `kubectl apply -f <file>` to activate it.

2. Create a ConfigMap named `outputkey` containing a single key named `REDIS_OUTPUT_KEY` with the value being a string where results from your function should be stored on Redis. To avoid conflicts with other students, you should store outputs from your function in a Redis key made of your student ID and the `-proj3-output` suffix, like `italocunha-proj3-output`.

> Data in a ConfigMap can also be loaded into a Pod as environment variables. This is the mechanism the runtime uses to get the information about where outputs from your function should be stored. You can generate this ConfigMap with the following command:
>
> ```
> # format: kubectl create configmap <name> --from-literal <key>=
> <value>
> kubectl create configmap outputkey \
>     --from-literal REDIS_OUTPUT_KEY=italocunha-proj3-output \
>     --output yaml
> ```

3. Deploy your serverless function by applying the provided deployment file to your Kubernetes namespace.

> The deployment file requires both the ConfigMaps defined in the previous steps. After editing all variables and create all needed ConfigMaps, the deployment can be started by `kubectl apply -f deployment.yaml`.

Task 2: Monitoring Dashboard

In this task, you will build a Kubernetes Pod (including the container image) to display the monitoring information computed by your serverless function. The following are a sequence of steps you should perform to complete this task:

1. Choose one dashboarding framework. There are many libraries for dashboarding data. In Python, the Plotly Dash and Streamlit frameworks are popular solutions, but students are free to choose how to build the dashboard (including frameworks not written in Python, like D3.js). When choosing your framework, consider that you will need to:

    1. Read data from Redis (step 3 below). A framework in a language that includes a Redis module is recommended (Python and JavaScript are OK).
    2. Extend the framework and package your extensions into a container to deploy in a Kubernetes Pod (step 4 below).

2. Implement code to show the metrics computed by your serverless function using the framework of your choice. Each framework should have documentation guiding you in this step. For example, the Plotly Dash library provides documentation examples and supports static or interative dashboards with dropdowns, checklists, input text, and others higher-level components.

> At this step, load some fake data into your dashboard, and run it directly on the virtual machine (without Docker or Kubernetes), to test that your use of the framework is correct. Create an SSH

> tunnel (see below) to be able to load the dashboard in a browser.

3. Read data from Redis into your dashboard. Remember from Task 1 that your serverless function's output is stored as JSON in Redis in a key given by your ID and the `-proj3-output` suffix. You can connect and access data on Redis from your chosen framework using the Redis packages. For example, the instructions for getting started with `redis-py` cover connection to and retrieval of a key in Redis. (See the section on Redis under Additional Information below.)

> At this step, repeat the previous test but now check that your dashboard is showing real data from Redis and that it updates periodically.

4. Package your dashboard in a Docker image, create a Kubernetes Deployment specification, and a Service specification. You should deploy your Pod on Kubernetes and expose your dashboard on a port given by the table below.

> In completing this step, test that your dashboard works from within the Docker image (without Kubernetes); run it using `docker run --publish hostPort:containerPort` and check that everything still works as expected. After testing on Docker, remove your container using `docker stop` and `docker rm`, and then proceed to test your Kubernetes Deployment and Service.
>
> This step overlaps strongly with Project 2. You can refer to your previous work and the documentation on Project 2. You can also use ArgoCD for this task if you want to automate deployment, but it is not required.

Task 3: Serverless Runtime

In this task you will build a container image to replace the runtime image provided by the instructors. You container image should be able to call serverless functions, passing any relevant parameters to the function and forwarding any returned values to their destinations.

In the runtime provided for this assignment, data is periodically read from Redis and passed in as parameters to the function. When the function returns, results are stored on Redis, where they can be later be read by the dashboard.

You should implement a compatible container runtime. Your container runtime must be able to replace the provided runtime and still operate with any function implemented for this assignment (assuming the function itself satisfied the integration requirements in Task 1).

Serverless Function Integration

You should study the provided deployment file to understand how the runtime accesses the Redis server, where it finds the user's Python module containing the `handler` function, where it reads data in Redis to forward to the function, and where it stores data `returned` by the function in Redis. Your runtime should use the same information to integrate with the Python modules provided by users.

Serverless Function Interface

You should study the interface for the `handler` function. Your runtime must provide the function with the same parameters (`input` and `context`), and accept the same type of result (a JSON-encodable dictionary).

> The input data is stored in Redis by a measurement program that you do not need to modify. It is sufficient to know that the measurement program stores data on Redis in JSON format, and that the JSON contains all the data described under "Data Input" above.
>
> The fields inside the `context` object should match those defined under "Context Information" above. You should generate a Python `class` with those fields; how your runtime obtains the necessary information to fill in each field is up to your implementation. Your runtime should ensure that changes to the `context.env` object persist between multiple calls to the `handler` function.
>
> Finally, your runtime should store JSON-encodable dictionaries returned by the `handler` function on Redis, on the key indicated by the user's `outputkey` ConfigMap (as defined in the deployment file).

Implementation Strategy

The minimum requirements to complete Task 3 are preparing a Python application that will periodically read data from Redis, call the user's `handler` function whenever new data arrives, and write and results back on Redis.

> The information needed to perform these operations are available from the deployment file and user ConfigMaps. In particular, the `handler` function is available from the user's module's source code, which you will need to `import` inside your Python application, and then call. For example, your Python application will need something along the lines of:
>
> ```
> import usermodule
> ...
> output = usermodule.handler(input, context)
> ```

Your Python application should check if the measurement data has changed since it was last read. If the data has changed, then your application should call the user's `handler` function and store the result in Redis.

> Note from the deployment file that the measurement data is stored in Redis in a key given by the `REDIS_INPUT_KEY` environment variable; similarly, results should be stored in a Redis key given by the `REDIS_OUTPUT_KEY` environment variable.

Consider providing flexibility in your runtime by preinstalling frequently used Python packages (e.g., `matplotlib`, `numpy`, `pandas`, `requests`, `sklearn`, `tqdm`) in your container image.

In this task, you should submit a Dockerfile, your Python application's source code, and any other files (like a `requirements.txt` file) required to build your container image.

Additional Information

Redis Module and CLI

In this project you will interact with Redis. In Python, the `redis` module can be used for this purpose. You can also use the redis CLI (`redis-cli`) to debug. The table below summarizes some useful commands to interact with Redis. These commands can also be executed from inside Python using the corresponding functions in the `redis` module.

Command	Description
`keys <pattern>`	List all keys following the given `<pattern>`
`set <key> <value>`	Set `<key>` to `<value>` in Redis
`get <key>`	Get value of `<key>`
`memory usage <key>`	Get memory usage of `<key>`

Redis is running on the VM on IP `192.168.121.189` and port `6379`; use these to connect to Redis to get and set key values.

Redis stores information as byte arrays, so you may need to call `encode` and `decode` in Python to convert the byte arrays into strings. When in doubt, always encode strings in `utf8`. Your serverless function should `return` a Python `dict` object, which will be saved in Redis as JSON. When reading data from Redis (for example, in your dashboard), you will need to parse the JSON to use the data in your framework. (In Python, the `json` module makes this task easy.)

Port Numbers for Dashboards

ID	Port
adalbertovieira	5101
arthurlima	5102
arthurmelo	5103
augustolessa	5104
barbararibeiro	5105
catarinapereira	5106
cunha	5107
fabiomorato	5108
felipecordeiro	5109
felipemingote	5110
flaviolucio	5111
gustavooliveira	5112
henriquefagundes	5113
jeffersonlopes	5114
joaocouto	5115
joaotrindade	5116
joaozica	5117
julioferreira	5118

ID	Port
lucasaguiar	5119
luissilva	5120
luizcouto	5121
marcossantos	5122
mateussilva	5123
matheussouza	5124
samuelalves	5125
viniciusramos	5126
wesleymaciel	5127
wesleyvieira	5128

What to Submit

You should submit:

1. For Task 1, submit your module's source code (containing the `handler` function), and two YAML files containing your ConfigMaps.
2. For Task 2, submit all files needed to build your container image and a short PDF containing screenshots of your dashboard and a discussion of the metrics monitored.
3. For Task 3, submit all files needed to build your container image and a PDF discussing the compatibility between your runtime and the runtime provided by the instructors.