WASP Graduate Course
Cloud Module Assignment

The aim of this assignment is to complement the theory you learnt with hands-on experience. In groups of 4 students, design, implement and test a scalable and fault-tolerant video conversion service on top of an Infrastructure-as-a-Service (IaaS) Cloud offering Virtual Machines (VMs).

In what follows, capitalized words are used as defined in RFC 2119.

Implement the **service**, the **workload generator** for it and a **monitoring tool** with the following requirements:

- The service MUST take input videos and convert them to an output video. The output format MAY either be hard-coded or received as a parameter for each input video.
- The service MUST be **scalable**. It MUST be able to scale up or down depending on its load, measured as the number of videos pending conversion. The exact choice of scale up and scale down strategy is left to you. However, please note that, due to oversubscription, the cloud environment you are provided with does not guarantee that adding one VM will lead to an overall increase in performance, nor that your VMs will have the same performance over time. Your scale up/down algorithm MUST handle such situations correctly.
- The service MUST be **robust**. It MUST be able to withstand some common failures of the IaaS cloud, such as VMs being killed.
- The workload generator MUST submit video conversion requests to the service and measure values such as response time[1] and queue length[2]. The workload MUST be generated in a closed system model[3] with two parameters: number of client converters, average time between conversion requests.
- The monitoring tool MUST interact with the IaaS cloud and periodically gather information such as number of VMs and CPU utilization of VMs. The tool's output MUST be optimized for plotting, e.g., CSV. You MAY include features in the monitoring tool to inject faults, e.g., randomly kill VMs.

Please provide the following deliverables by email:

1. **Report on architecture**: Before writing any code, report the architecture of the application, consisting of components, exposed interfaces and technology stack employed for each component. Also describe how your application will be scalable, when will it scale up and down and how you will ensure it is robust. The report will serve as a milestone to ensure the requirements of the application can be implemented. **Deadline: May 26**.

[1] The amount of time elapsed from the client submitting an input video for conversion to the full output video being ready.
[2] The number of input videos submitted that do not have an output video ready.
[3] Please refer to Schroeder et al, "Open Versus Closed: A Cautionary Tale", Section 2.

2. **Report on scalability and robustness**: After having completed the implementation, use the workload generator and monitoring tool to perform experiments that show that your service is both scalable and robust. Make sure that your experiments highlight at least one scale-up, scale-down and failure event. Report plots that show how these events influence the metrics shown by the workload generator -- response time and queue length -- and the monitoring tool -- number of VMs and their CPU utilization. **Deadline: June 21.**

We expect the effective work not to exceed 40h per group (120 or 160 person-hours). We left generous deadline to allow you to schedule around your other commitments, but **we strongly encourage you to submit the deliverables earlier**.

Please send the deliverables to: Cristian Klein <cklein@cs.umu.se>, Ewnetu Bayuh <ewnetu@cs.umu.se>, and Olumuyiwa Ibidunmoye <muyi@cs.umu.se>.

We suggest the following technology stack, but you are free to use whatever tools you see fit:

- Ansible for managing your VMs -- e.g., having the right packages installed -- and deploying code on them.
- REST API between the workload generator and the service. Please read this blog regarding how to design a REST API for long-running operations.
- Mencoder for video conversion.
- Python Flask for implementing REST servers.
- Python requests for making REST requests, either to the service or to the IaaS cloud (OpenStack).
- Java Jersey to implement REST clients and servers.
- etcd for distributed configuration management. (When and how to use it?)
- RabbitMQ for maintaining a distributed queue.
- Python novaclient for interacting with OpenStack.
- 4K Video Samples for some input videos.
- nginx for load balancing.