Środowiska udostępniania usług

Implementation of Network Services

# Project Kopf-15 - Kubernetes Operators Framework

*Authors:*
*Paweł Hanusik*
*Tomasz Koszarek*
*Wojciech Piechaczek*
*Tomasz Nal*

*Year, Group:*
*2023, Friday 13:00*

# 1. Introduction

This section contains a short introduction to the basic findings of the project. Described tools are Kubernetes and Kubernetes Operators with their usage.

### 1.1 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Kubernetes utilizes virtualization. It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application.

Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, the share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Deploying Kubernetes means setting up a cluster consisting of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault tolerance and high availability. [1]

### 1.2 Operators

Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.

Kubernetes is designed for automation. Out of the box, the user gets lots of built-in automation from the core of Kubernetes. You can use Kubernetes to automate deploying and running workloads, and you can automate how Kubernetes does that.

Kubernetes' operator pattern concept lets the user extend the cluster's behavior without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.

Some of the things that one can use an operator to automate include:

- deploying an application on demand
- taking and restoring backups of that application's state
- handling upgrades of the application code alongside related changes such as database schemas or extra configuration settings
- publishing a Service to applications that don't support Kubernetes APIs to discover them
- simulating failure in all or part of your cluster to test its resilience
- choosing a leader for a distributed application without an internal member election process

## 2. Theoretical background/technology stack

This section describes what Kopf is and how one can use it.
Also, the technology stack for the project is presented, including programming languages, frameworks, and other tools.

### 2.1. Theoretical background

Kopf [2] is a Python framework that allows the building of Kubernetes operators, which are custom controllers that can automate the deployment and management of Kubernetes resources. By using Kopf with Kubernetes, developers can automate common tasks such as scaling, updating, and monitoring their applications, making it easier to manage large, distributed systems.
With Kopf, developers can define the desired state of their applications and let the operator manage the actual state, without manual intervention. Kopf also provides a number of advanced features, such as handling resource dependencies, reacting to resource changes (on creation, on deletion, etc.), and managing complex failure scenarios. Overall, Kopf can significantly simplify the management of complex applications in Kubernetes and provide developers with a powerful tool for building resilient and scalable systems.

### 2.2. Technology stack

- Kubernetes/kubectl [1]- An open-source container orchestration platform, used to deploy and manage containerized applications at scale. Kubernetes has become a de facto standard for enterprise infrastructure management, especially for microservice-based infrastructures.
  kubectl is a Kubernetes command-line tool, it allows running commands against Kubernetes clusters.
- minikube [5] - A lightweight tool that allows developers to run a single-node Kubernetes cluster on their local machines for testing and development purposes.
- Kopf [2] is a framework to build Kubernetes operators in Python.

- MS Azure - A cloud computing platform that offers a wide range of services, including virtual machines, storage, databases, and analytics. One of the services from Azure is the Azure Kubernetes Service (AKS) [6] - used as a hosted Kubernetes service in the cloud.
- Yaml (YAML Ain't Markup Language) is z human-readable data serialization language that is often used for configuration files, data exchange, and API definitions. It is designed to be easy to read and write, making it popular for use in settings where both humans and machines need to read and interpret the same data. Yaml is used for defining and configuring Kubernetes objects.
- Python SDK for Kubernetes [3] - A set of libraries and tools that allow developers to interact with Kubernetes clusters using the Python language. The Python Kubernetes SDK is used to automate tasks such as deploying, managing, and scaling containerized applications on Kubernetes.

# 3. Case study concept description

This section describes our proposed case study and demo which will showcase a distributed RSA bruteforcer.

## 3.1. Problem domain

We would like to create a RSA cipher brute forcer using distributed computing. Since the problem underlying RSA encryption - large number factorization - cannot be solved efficiently without a quantum computer, the only way to speed up this process is to leverage parallelism and high computing power. Even when parallelizing the workload, the RSA cipher has been designed to take an inconceivable amount of time to break. Therefore we will only attempt breaking a variant of this algorithm with shorter keys compared to the ones used presently to secure real data.

## 3.2. Concept description

To accomplish this parallelization we aim to create multiple workers in Kubernetes which will each work on a part of the solution. These workers will be ultimately deployed in the Azure cloud where we will evaluate the final solution. To avoid having to create worker instances by hand, we plan to create a custom Kubernetes resource that will create the workers and any of their dependencies automatically after being deployed. As an extra challenge, we have decided to make the workers stateful, with a need for backups and state preservation. We understand this is not a very practical solution (Kubernetes is more suited for stateless applications), but this hurdle will allow us to showcase the strengths of Kopf and Kubernetes operators in general.

## 3.3. Use of Kopf

This case study concept necessitates the use of an operator to implement the required functionality for the custom resource and to manage the state of the workers. Our operator will also perform the task of periodically backing up the progress of the RSA breaker, for the purpose of disaster recovery.  The operator shall be written in the Python language using Kopf. This study has been designed to highlight the many features provided by Kopf, such as event handlers, state handling, and daemon jobs.

### 3.4. Resulting artifacts and evaluation

The final solution will be an example deployment which will deploy an operator, a custom resource definition, multiple workers, and other necessary resources. See chapter 4. for a description of all the necessary resources we identified for the final demo. The solution shall be evaluated in the cloud in terms of the correctness of behavior, performance, and fault tolerance. This will be achieved by simulating real disaster scenarios, e.g. removing a node while the calculations are in progress.

# 4. Solution architecture

This section contains a brief definition of all the resources used in our project, and the relations between them.

### 4.1. Worker node

The worker is the fundamental component of the system. It comprises a program code that aims to factorize a given RSA public key by utilizing numbers from a supplied range.

### 4.2. Master node

The master node's primary function is to receive user requests, which take the form of valid RSA encryption keys. Additionally, users may have the option to exercise fine-grained control over the cracking phase, such as setting the desired number of nodes. The master node is responsible for preparing and dispatching tasks to the workers, as well as keeping track of the range of numbers already attempted and their corresponding outcomes.

### 4.3. Service

To ensure accessibility to the master node, we will use the Kubernetes Service [7], which allows for the exposure of a network application running within our Kubernetes solution to the external world.

### 4.4. Kopf operator

The Kopf operator is a key component of this architecture, responsible for monitoring all worker nodes, tracking their changes and progress, and responding to failures by recovering the achieved state of the computation and reassigning it to new nodes.

### 4.5. Persistent Volume Claim
To maintain consistency within the system and avoid redundant work by the workers, we are leveraging the Kubernetes storage solution - Persistent Volume Claim [9]. This solution is mounted by each worker upon its creation and serves the purpose of persisting the state of the computation in case of a worker failure. In the event of such a failure, the operator detects its occurrence and reads the saved state from the storage. The operator then communicates with the master node to reassign the work to another worker.

### 4.6. Custom Resource Definition
Our solution is defined using a Kubernetes functionality called Custom Resource Definition [8]. This feature extends the Kubernetes API, enabling the description of multiple objects and their respective relations, which can be deployed to the cluster based on those instructions. Once a custom resource is installed, its objects can be created and accessed using kubectl, just like built-in resources such as Pods.

# 5. Environment configuration description

This project can be run in two distinct environments: in the cloud or locally.

### 5.1. Cloud environment
We're using Microsoft Azure as our cloud provider. Our cloud environment makes use of the Azure Kubernetes Service (AKS) provided by Microsoft.  AKS  offers a quick way to start developing and deploying cloud-native apps in Azure. The Azure subscription provided to us only allows us to provision a Kubernets cluster limited to two nodes, due to limitations imposed on the subscription by Microsoft.

Instruction on cloud deployment are provided in chapter 7 of this documentation.

### 5.2. Local environment
In order to run and develop the application locally we made use of minikube. Minikube allows us to create a single-node Kubernetes environment locally.

Instruction on local minikube deployment are provided in chapter 7 of this documentation.

# 6. Installation method

The only prerequisites for running this project are:

- A valid Kubernetes environment. This means having a local minikube installation or using a cloud provider to provision a kubernetes environment.
- An up-to-date installation of git, to clone the project files.

The project can be cloned with:
*git clone https://github.com/Tomn0/SUU-Kopf.git*

There is no need to compile or build this project, as all artifacts have been pre-compiled and published on DockerHub - a public Docker image-sharing platform.

After cloning the project the user can manually apply the .yaml files found in the "deployments" folder, or use the step-by-step instructions provided in Chapter 7 of this documentation to run our demo installation.

# 7. How to reproduce - step by step

In this section, we provide step-by-step instructions for getting our infrastructure and deployment running. We provide the instructions first for a cloud deployment, then a local deployment and lastly we discuss other deployment options.

## 7.1. Cloud environment in Azure

Prerequisites: a valid Azure subscription and AZ CLI installed.
Use these instructions when deploying to the Azure cloud.

### 7.1.1 Infrastructure as code approach

We have made use of the Infrastructure as Code approach to automate the process of creating our Azure infrastructure. Azure uses a custom language called Bicep to define deployments of Azure resources. The Azure command line utility (AZ CLI) contains commands which create deployments based on Bicep definitions. A sample of our Bicep definition is provided below:

```
resource aks
'Microsoft.ContainerService/managedClusters@2022-05-02-preview' =
{
  name: clusterName
  location: location
  properties: {
    dnsPrefix: dnsPrefix
    agentPoolProfiles: [
      {
        name: 'agentpool'
        osType: 'Linux'
        mode: 'System'
      }
    ]
  }
}
```

On top of a full infrastructure definition in Bicep we also provide bash and PowerShell scripts that deploy the infrastructure in the cloud.

### 7.1.2 Step-by-step instructions for Azure Cloud

Deploying the infrastructure:

1. In your terminal authenticate to your Azure account: *az login*
2. In case you use multiple Azure subscriptions set the desired one with: *az account set --subscription YOUR_SUBSCRIPTION*
3. Deploy the infrastructure with:
   - Bash: *./infrastructure/scripts/bash/deploy.sh*
   - Powershell: *./infrastructure/scripts/powershell/deploy.ps1*

Deploying the example deployment:

1. Apply the operator: *kubectl apply -f .\deployments\operator\*
2. Apply the custom resource definition: *kubectl apply -f .\deployments\crd\*
3. Apply the example deployment: *kubectl apply -f .\deployments\example*

### 7.1.3 Azure Cloud cleanup

Deleting the deployment:

1. Delete the resources: *kubectl delete -f .\deployments\example*
2. Delete the custom resource definition: *kubectl delete -f .\deployments\crd*
3. Delete the operator: *kubectl delete -f .\deployments\operator\*

Deleting the infrastructure:
1. Run:
   - Bash: *./infrastructure/scripts/bash/teardown.sh*
   - Powershell: *./infrastructure/scripts/powershell/teardown.ps1*
2. **Please note**: Unless explicitly disabled, Azure creates a Network Watcher resource when deploying a Kubernetes Cluster. This will add additional costs

and is not removed by the script above. If you want to delete the Network Watcher (recommended unless you're using it for other projects) use:

- Bash: *./infrastructure/scripts/bash/teardown-nw.sh*
- Powershell: *./infrastructure/scripts/powershell/teardown-nw.ps1*

## 7.2. Local environment with Minikube

Prerequisites: Minikube.
Use these instructions in local deployments.

### 7.2.1 Step-by-step instructions for Minikube

1. Start minikube: *minikube start*
2. Apply the operator: *kubectl apply -f .\deployments\operator\*
3. Apply the custom resource definition: *kubectl apply -f .\deployments\crd\*
4. Apply the example deployment: *kubectl apply -f .\deployments\example\*

Now you can use *kubectl* or *minikube dashboard* to monitor the deployment.

### 7.1.3 Minikube cleanup

Deleting the deployment:

1. Delete the resources: *kubectl delete -f .\deployments\example*
2. Delete the custom resource definition: *kubectl delete -f .\deployments\crd*
3. Delete the operator: *kubectl delete -f .\deployments\operator\*
4. Delete minikube: *minikube deletewn-nw.ps1*

## 7.3. Other deployment options

### 7.3.1 Deploying for other cloud providers

Although our infrastructure as code definitions are Azure-specific, the solution as a whole will work with any Kubernetes provider. If you have a different cloud provider available, you may use the instructions provided for Minikube, and ignore the initial *minikube start* command to deploy the project.

### 7.3.2 Deploying locally for development purposes

The above instructions were designed with deploying the final product in mind. Since the project is quite complex it’s difficult to actively develop it in an environment similar to the ones described above.

To make development easier we have developed specific instructions for comfortable local development, based on Minikube. These instructions can be found in the README file on our github page.

## 8. Demo deployment steps:

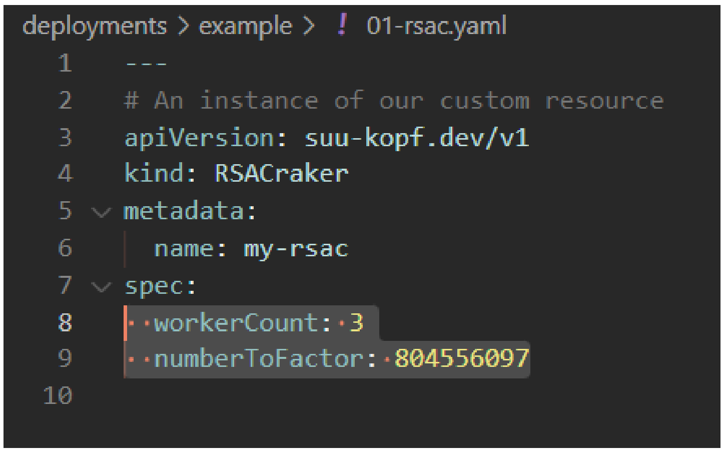
### 8.1. Configuration set-up

Run Docker
In the PowerShell terminal run commands `minikube start` and `minikube dashboard` - this will create a local Kubernetes environment as well as open a web dashboard that gives more visibility into what is happening in the environment.

Open the project
In Powershell invoke the minikube shell: `& minikube -p minikube docker-env --shell powershell | Invoke-Expression`

### 8.2. Data preparation

Set the calculation parameters inside the file 01-rsac.yaml


deployments > example > ! 01-rsac.yaml

```yaml
---
# An instance of our custom resource
apiVersion: suu-kopf.dev/v1
kind: RSACraker
metadata:
  name: my-rsac
spec:
  workerCount: 3
  numberToFactor: 804556097

```

### 8.3. Execution procedure

Build the Docker images using the commands from README.
Apply the images using the commands from README.

To change the number of workers dynamically:
In order to set up the calculations modify the `workerCount` and parameters in the 01-rsac.yaml file.

### 8.4. Dynamic workers scaling

To change the number of workers dynamically:

1. Run: `kubectl edit RSACracker my-rsac`
2. Write the desired number of workers
3. Save and close the file

### 8.5. Fault tolerance

From the dashboard kill one of the workers.
The operator is tasked with maintaining the preset number of workers so it will create a new worker to continue the killed worker's task.

### 8.6. Communicating with the workers

The workers are saving the progress of their work inside the PVC storage. We can directly examine those files or make use of Flask endpoints that make accessible the calculations from there.

```
root@master:/# curl http://localhost:8080/progress
{
  "progress_all": 94526,
  "progress_done": 84555,
  "solution": 15581
}
```

### 8.7. Results presentation

To get the progress open the shell inside the master pod and type:
curl http://localhost:8080/progress

```
root@master:/# curl http://localhost:8080/progress
{
  "progress_all": 94526,
  "progress_done": 84555,
  "solution": 15581
}
```

## 9. Summary – conclusions

As part of this project, we have created an operator which facilitates a distributed RSA brute forcer. We make use of Kopf - the pythonic operator framework - to create a working operator, capable of reacting to environment changes, and creating custom resources.

This project provided us with a great deal of knowledge and experience with the Kopf framework. Through its use, we have learned about its capabilities such as:

- Reacting to configuration changes
- Detecting creation and deletion events in the Kubernetes environment.
- Patching Kubernetes resources with modified data
- Making comparisons between original and updated data, when presented with a configuration change, and reacting accordingly
- Retrying failed operations with a robust and configurable retry and backoff system

We have found the framework to be well-documented and comparatively easy to use. The syntax of Kopf's API makes heavy use of Python decorators which effectively reduces boilerplate and speeds up development. The community around Kopf, however, is still rather small, which makes it hard to find resources and ask questions.

Additionally, we have learned about the following technologies unrelated to Kopf:

- Minikube as a local Kubernetes environment
- The Kubernetes Python API
- Kubernetes volumes and storage claims
- The Bicep language for creating infrastructure-as-code scripts for the Azure cloud.

The team considered this project a success, and can wholeheartedly recommend the usage of the Kopf framework for operator development.

## 10. References

[1] Kubernetes documentation kubernetes.io/docs
[2] Kopf documentation https://kopf.readthedocs.io/en/stable/concepts/
[3] Kubernetes client for Python https://pypi.org/project/kubernetes/#files
[4] GitHub repository for the project https://github.com/Tomn0/SUU-Kopf
[5] minikube documentation https://minikube.sigs.k8s.io/docs/start/
[6] Azure Kubernetes Service https://azure.microsoft.com/en-us/products/kubernetes-service
[7] Kubernetes Service https://kubernetes.io/docs/concepts/services-networking/service/
[8] Custom Resource Definition https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[9] Persistent Volumne Claim https://kubernetes.io/docs/concepts/storage/persistent-volumes/