Bachelor Thesis

August 17, 2020

A Cloud Framework for Polyglot Parallel Genetic Algorithms

Janik Lüchinger

of Oberriet, Switzerland (17-704-339)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Bachelor Thesis

A Cloud Framework for Polyglot Parallel Genetic Algorithms

Janik Lüchinger

University of Zurich^UZH

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis
Author: Janik Lüchinger, janik.luechinger@uzh.ch
Project period: 21.02.2020 - 17.08.2020

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I would like to thank my colleagues and friends Constantin Beer, Vanessa Rüegg, and Damaris Schmid for proofreading my work and providing some valuable inputs. I also wish to thank my family for supporting my every decision during my studies and for helping out as much as possible. Finally, I want to express my heartfelt thanks to my study group, Christoph Fässler and Damaris Schmid. Together we were sweating blood, but we also helped each other through the tougher times. I wish you all the best for your future.

Abstract

Genetic Algorithms are a potent tool when computing an exact solution for a problem is too expensive, but a near-optimal approximation can be sufficient instead. Most Genetic Algorithms are sequential programs that are prone to scalability issues. Increasing their performance is possible by executing resource-intensive steps in parallel, therefore reducing the required computation time. Some previous proposals for cloud-based Genetic Algorithm distribution already provided frameworks exploiting well-known parallelization techniques. We devised a new, more flexible framework. We propose *PGAcloud*, a cloud framework capable of including and executing polyglot, i.e., multi-language, Genetic Algorithms and deploying them to a prepared cloud environment. Developers of Genetic Algorithms can include their custom implementations into a wrapping software container, effectively deploying a local algorithm to the cloud, without worrying about the underlying implementation details of the framework. Deploying a Genetic Algorithm to the cloud for parallelization makes it a Parallel Genetic Algorithm. Allowing developers to include any code into the framework directly makes our proposed framework very flexible. *PGAcloud* employs an easily scalable architecture and takes care of cloud orchestration, load balancing, provisioning, and deployment of the required software containers. After any adjustments to the provided Parallel Genetic Algorithm configuration template, the user simply needs to execute the desired commands from the local client's command-line interface and point out the configuration file to be used. By basing the main capabilities of any Parallel Genetic Algorithm computation on a user-defined configuration file, we keep the possibilities for future additions as versatile as possible.

Zusammenfassung

Genetische Algorithmen sind ein mächtiges Werkzeug zur Problemlösung, wenn die Berechnung einer exakten Lösung zu teuer ist, eine nahezu optimale Annäherung jeodch ausreicht. Die meisten Genetische Algorithmen sind sequenzielle Programme und daher anfällig für Skalierungs-Probleme. Ihre Leistung kann jedoch gesteigert werden, indem gewisse Ressourcen-intensive Schritte parallel ausgeführt werden, wodurch die benötigte Rechenzeit reduziert wird. Einige frühere Vorschläge für die Cloud-basierte Verteilung Genetischer Algorithmen haben bereits Frameworks angeboten, welche bekannte Techniken zur Parallelisierung nutzen. Wir entwickelten ein neues, flexibleres Framework. Wir schlagen *PGAcloud* vor, ein Cloud-Framework, welches polyglotte, d.h. mehrsprachige Genetische Algorithmen einbinden und ausführen sowie sie auf eine vorbereitete Cloud-Umgebung verteilen kann. Entwickler von Genetischen Algorithmen können ihre problemspezifischen Implementationen in einen Software Container einbinden, was einen eigentlich lokalen Algorithmus in der Cloud bereitstellt, ohne sich um die darunterliegenden Implementierungs-Details des Frameworks zu kümmern. Die Bereitstellung eines Genetischen Algorithmus zur Parallelisierung in der Cloud macht daraus einen Parallelen Genetischen Algorithmus. Den Entwicklern zu ermöglichen, Code jeglicher Art direkt in das Framework einzubinden, macht unser vorgeschlagenes Framework sehr flexibel. *PGAcloud* verwendet eine einfach skalierbare Architektur und kümmert sich um Cloud Orchestrierung, Lastverteilung, Vorbereitung und Aufschaltung der benötigten Software Container. Nach allfälligen Anpassungen an der Konfigurationsvorlage des Parallelen Genetischen Algorithmus muss der Nutzer lediglich die gewünschten Befehle im Kommandozeilen-Interface des lokalen Clients ausführen und ihn auf die zu verwendende Konfigurationsdatei hinweisen. Indem wir die Hauptfähigkeiten einer jeden Berechnung eines Parallelen Genetischen Algorithmus auf einer nutzerdefinierten Konfigurationsdatei basieren, halten wir die Möglichkeiten für zukünftige Ergänzungen so vielseitig wie möglich.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Genetic Algorithms (GAs) are an impressive tool for solving problems where directly computing an optimal solution to a complex problem is too expensive, but a near-optimal approximation is sufficient to solve it [2].

Although glorified by academic research in laboratory conditions, GAs usually are programs of sequential nature, making them prone to scalability issues. Luckily, it is also in their nature to be easily parallelizable, increasing their performance by reducing the required computation time [3, 4]. Algorithms implementing such parallelization approaches (like the “Master-Slave,” “Island,” or “Grid” model) are known in scientific literature as forms of Parallel Genetic Algorithms, or PGAs for short [3,5].

Considering that computing the best approximation in a parallelized way may consume most of the computation power available on a local machine, we can further improve the PGAs’ performance by distributing them in the cloud and performing the computation on multiple machines.

Previous proposals for cloud-based GA distribution exploited well-known parallelization techniques, and some of them provided frameworks to support the development and deployment of distributed Genetic Algorithms [2].

With *PGAcloud*, this work aims to add a new, more flexible framework. Its main contributions include supporting PGA developers in deploying their local algorithms to their preferred cloud environment by allowing them to directly include any code into the framework without worrying about the underlying implementation details. For example, this characteristic facilitates the collaboration of multiple developers together to solve the same problem by combining each of their respective implementations of different operators. Furthermore, supporting the integration of polyglot programs (i.e., a program written in or combining multiple programming languages) can be of great value when a developer finds the source code of an existing implementation for some genetic operator that is written in any programming language and wants to integrate it into his own approach. Features like these make our proposed framework very flexible and developer-friendly while also employing an easily scalable architecture at the same time.

PGAcloud takes care of cloud orchestration and load balancing, provisioning, and deployment of the required software containers. By basing the main capabilities of any PGA computation on a user-defined configuration file, we keep the possibilities for future additions (like dynamically changing properties or monitoring the PGA at runtime) as generic and easy as possible.

Thesis Structure. The rest of this work is structured as follows: In the upcoming Chapter 2, we explain the background and motivation for developing the *PGAcloud* framework, where Sections 2.1 and 2.2 focus on relevant concepts of previous framework propositions. In the subsequent Chapter 3, we propose the *PGAcloud* framework and explain its architecture, followed by Chapter 4, which explains the prototype we devised. Section 4.1 lists the details of the technologies we used. Then, as part of Sections 4.2 and 4.3, we illustrate the concepts of the different framework components and shine light on some limitations and pending refinements of *PGAcloud*. Section 4.4 is all about our example implementation for solving the Knapsack problem. Finally, in Chapter 5, we state our conclusions, summarize our propositions, and elaborate on possible extensions and future work.

Chapter 2

Related Work

Genetic Algorithms (GAs) are a potent tool for solving problems where directly computing an optimal or exact solution is too expensive, but a near-optimal approximation of such an ideal solution (being acquired over multiple iterations) is sufficient to solve the problem at hand [2]. The process imitates the biological concept of evolution: evolving an initial population of individuals (possible solutions to the problem) over many generations. Within each generation, the individuals are subject to various genetic operations like mating (selection and crossover), mutation, or the classic "survival of the fittest" (elitism and survival selection). Most importantly, "the heart of GAs is the fitness function" [2], which computes an individual's fitness and maps it to a numeric value, finally allowing the function to evaluate or classify each individual within the entire population [2].

Usually, GAs are programs of sequential nature, making them prone to scalability issues, possibly preventing the application of GAs in an industrial scenario [3]. Parallelization of specific time-consuming steps is a possible way of massively increasing their performance in terms of required computation time [4], providing a solution to the scalability problem. According to Salza and Ferrucci [2], the "high cost of parallel architectures and infrastructures and their management" is arguably one reason why parallelized execution has not found a more prominent application so far.

Nonetheless, Genetic Algorithms have previously been parallelized on multi-core (i.e., CPUs) as well as many-core (i.e., GPUs) systems – successfully increasing their performance and effectiveness [6, 7]. However, as stated above, applying such solutions is often expensive. Their achievable degree of parallelization is directly related to – and limited by – "the number of multiple computational units available on the hardware" [2]. In contrast, to free the computation from environmental restrictions, network-based technologies like cloud or grid computing are scalable without limitation – at least in theory [8,9]. Notably, cloud computing serves as a more affordable solution. It addresses both the hardware limitation and the cost aspect: allocating a cluster of as many nodes as required is possible in a short time. The main advantage is that one would not have to invest in expensive hardware, which would have to be maintained and managed [2,10,11].

Parallelization Models. In their work, Luque and Alba [5] proposed three forms of parallelization models. For instance, the GAs characteristic of being based on an evolutionary population allows evaluating each individual's fitness in parallel, resulting in the global parallelization model (also called the "Master-Slave" model). In this model, the "Master" node is responsible for the coordination of the PGA computation. It maintains the population and performs genetic operations like selection, crossover, or mutation on each individual. The parallelization occurs at the fitness evaluation level, when the "Master" distributes the fitness computations to several "Slave" nodes and simply collects their results. Besides that, there are other ways of exploiting parallelization, e.g., when performing genetic operations as part of the coarse-grained model (also known as the "Island" model), to ultimately raise the next generation of possible solutions. The coarse-grained model applies parallelization at the population level. More precisely, it splits the complete PGA computation into multiple subpopulations in a "Master-Slave" fashion. These subpopulations are called "islands," hence the name "Island" model. To ensure genetic diversity, migration of individuals between the islands may occur statically or dynamically [12]. A third possibility is to combine the two strategies above to produce the fine-grained model (the "Grid" model). This model distributes the individuals in a multidimensional grid-like fashion. Like in the "Master-Slave" model, genetic operations are limited to the maintained subpopulation, i.e., one individual in the grid and its immediate neighbors. Also, migration may occur between subpopulations during selection. The algorithms implementing these approaches are known in scientific literature as forms of Parallel Genetic Algorithms, or PGAs for short [3,5]. The three mentioned PGA models are not the only ones [12,13], but the most commonly used.

From the PGA models explained above, this work focusses on the "Master-Slave" model. Since many previous works proposed new aspects or approaches for this model, we included the related work for the concepts that influenced the design and development of our *PGAcloud* framework. The presented concepts are mainly centered around containerization and internal communication, general PGA architecture or targeted audience and their different roles.

2.1 Parallel Frameworks for Genetic Algorithms

In their works, Di Geronimo et al. [14], and later Di Martino et al. [15], suggested using Hadoop MapReduce to relieve programmers from the underlying implementation issues for orchestrating a distributed computation, mostly since MapReduce is supported by most cloud providers. Some first results of Di Martino et al. showed that the cloud environment could massively outperform a local server. This idea was then picked up by Salza, Ferrucci, and Sarro when they proposed *elephant56*[1], which was the "first publicly available framework based on Hadoop MapReduce" [3].

When integrating with *elephant56*, a developer is expected to extend specific classes defined within the "user package" and customize them according to the problem to solve. These classes are related to the different phases of the job, the model of individuals, and the genetic operators [3].

Their proposed framework had several exciting concepts, three of which also influenced the architecture of the framework resulting from this work: with *elephant56* [3], Salza et al. mentioned the usage of an initial population, which would be sent to the fitness evaluation before starting the actual GA computation. They did not explicitly state whether a user provided this population, or the framework randomly generated it. However, it is clear that both options are of value to any user and thus were provided in our framework *PGAcloud*. In addition to the initial population, Salza et al. [3] introduced the notion of having global properties (e.g., distinct commands or names of message queues) being filled and distributed to the operators by the "master node." Providing properties to the operators makes way to two scenarios: firstly, a user might be able to dynamically change these properties at runtime instead of statically providing them in the beginning. The "master node" would be responsible for forwarding any changes to the corresponding genetic operators. Secondly, allowing to provide any number of unforeseen, user-defined properties eases flexibility and accessibility of including different implementations of operators, possibly extending or completely overriding the provided default implementation of the framework. The last relevant concept by Salza et al. is the already mentioned "master node." With *elephant56*, they proposed having a single class (or node, respectively) in charge of the entire PGA workflow (the "Driver"), which is similar to the *PGAcloud* implementation (the "Runner," see Section 4.2

---

[1] https://github.com/pasqualesalza/elephant56

on page 18). Also, the Driver is "the linking point between core and user layers and primary interface with the developer" [3], facilitating a single point of contact.

As indicated by the core level, merely having a Driver class is not everything a distributed PGA requires to function. It must also employ some means of storing and distributing data across multiple nodes or containers. As one possible approach towards this, Garcia-Valdez et al. implemented the *EvoSpace Model* [16], which consists of two main components: some repository (e.g., a database) storing the population resulting from genetic evolution, and multiple remote workers, which implement and execute the actual evolutionary process. Considering that there are several such worker nodes in a cloud application that need "to be created and destroyed in seconds to guarantee the reliability and scalability of the entire system" [2], the traditional hypervisor-based virtualization approach for cloud computation becomes a limiting factor. Such a system generally operates on the hardware level – contrary to container-based virtualization, which performs its computations on the operating system level. Because of that, container-based virtualization can provide a rather lightweight virtual environment, namely the software container. Software containers bundle related processes and isolate them from other processes, containers, or hardware on the host machine [2]. Every container accesses the same shared kernel of the host system and can be much smaller and more lightweight than an entire virtualized operating system [17]. With this isolation in place, a containerized process can only see resources within this same container, meaning that the underlying host network is the only available form of communication [2].

As a direct result, the increased parallel communication load on the network makes it a bottleneck, limiting such a system's performance. Consequently, it is not trivial to run distributed PGAs in an on-demand fashion. The first time a genetic algorithm had been containerized using Docker to be deployed in the cloud was with the work of [10]. Its authors highlighted that their containerized approach allowed them to benefit from allocating resources on-demand, which is "one of the most attractive features of cloud computing" [18].

Salza and Ferrucci adopted this idea when they designed and implemented *AMQPGA*[2], a novel distribution approach for cloud-based Genetic Algorithms [2]. It was part of their key goals to keep the communication cost as low as possible, while simultaneously assisting developers at developing and deploying their algorithms. Their proposition implements the "Master-Slave" model and exploits container-based cloud orchestration, relying on communication via message queues. With their container-based approach, they were able to "surpass the limitations of the number of machines the cloud providers usually impose upon their users" [2]. The resulting infrastructure is definable as "multi-cloud," i.e., the infrastructure can integrate into the same application multiple nodes allocated by different cloud providers [2]. This flexibility was made possible through communication relying on network traffic.

Di Martino et al. [15] were among the first to focus on executing GAs in the cloud. Not only did they suggest using Hadoop MapReduce, but they also proposed the parallelization of the GA at three different levels: fitness evaluation, population, and individual. Their implementation, similar to the more recent *AMQPGA* of Salza et al. [2], chose to parallelize the computation on the fitness evaluation level, i.e., to evaluate the fitness of individuals in parallel. Di Martino et al. reported significant overhead due to the resulting communication in the network [15, 18]. The communication overhead in transmitting data likely was additionally increased by choosing "the rather verbose JSON data-interchange format" [18].

---

[2] https://github.com/pasqualesalza/amqpga

2.2 User Roles

We already established that our framework intended to support developers in their practice of developing and deploying GAs. To explain how we do so, we must first introduce two concepts: the first is the concept of code integration, where a software container can include the code of a different algorithm. In this context, the second concept is understanding the targeted audience of the framework, splitting it into groups, and then designing the software to match their respective roles. Similar concepts have been mentioned and implemented by Salza et al. in their work on *cCube*. Although their work addresses Evolutionary Machine Learning and the intended goal was different, *cCube* inspired the same user identification as was ultimately employed by *PGAcloud*.

In 2017, Salza et al. presented *cCube*[3] [1], an architecture supporting its users in developing and deploying algorithms for Evolutionary Machine Learning (EML) to the cloud. The most notable characteristic of *cCube* is its microservices architecture, where user code can be integrated into software containers without having to worry about any underlying implementation details. Doing so allows EML developers to benefit from cloud computing and on-demand allocation of resources, without being locked in on a particular cloud provider (Amazon Web Services[4], Google Cloud Platform[5], OpenStack[6], Etc.) [1]. "Collectively, *cCube*'s services are minimally centralized and managed by an *orchestrator*" [1], whose implementation is generalized and independent of the chosen cloud provider(s).

Their work introduced three different roles related to the *cCube* framework: EML algorithm developers, EML end-users (technically unfamiliar with EML insights), and the *cCube* engineers [2].

The *cCube* engineers' responsibility is to expand and maintain the openly available source code and provide the most recent functionality to interested EML developers [2].

The role of EML developers could also include researchers focusing on EML algorithm design or development. In particular, developers could deploy EML algorithm implementations in different programming languages. In doing so, they would not need to inject any *cCube* related code into their EML algorithm to enable cloud compatibility or capabilities. Instead, *cCube* is injected in the EML algorithm's container, where the container runs it as a daemon to manage inter-container communication within the system. Developers are only expected to adhere to specifications of the provided input/output interface: to successfully integrate with *cCube*, developers must customize a configuration template from the *cCube* repository and provide it to the framework's code according to instructions. Their role then changes as they become end-users, starting the client locally on their machine. Provided authorization keys for the cloud environment are sensitive data, thus kept local and secure. After establishing a secure connection, the *cCube* client starts the Docker application in the cloud, making it available for user interactions [1].

Regarding Docker containers, Salza et al. [1] write in their work on *cCube*:

> Using Docker, we extended the development capability to allow the developer to include source code and/or to define the algorithm's execution environment, i.e., every component required for learner execution in any programming language or technology, without requiring a manual development intervention. The interaction interface is kept flexible by defining a wrapping interface. Therefore, the only information required is the path and instruction on how to execute the [EML algorithm]. The developer does not need to make the source code aware of *cCube*'s functionality or parallel computation. Therefore, *any* algorithm can be executed in *cCube*. Once the

---

[3] https://github.com/ccube-eml

[4] https://aws.amazon.com

[5] https://cloud.google.com

[6] https://www.openstack.org

container is defined, the developer only needs to build and distribute it on a *Docker Registry* repository to be downloaded, executed and replicated on demand.

End-users like multi-algorithm EML application managers or non-EML literates (i.e., end-users without the technical EML knowledge) usually treat *cCube* as a black box when trying to execute large-scale EML algorithms in the cloud. Using provided cloud account credentials, *cCube* allocates, provisions, and distributes the computational units. The user only needs to provide a valid configuration by customizing the template. *cCube* then forwards the request to the cloud provider(s) and orchestrates the deployed containers [1].

When the user has submitted a new job to the cluster, *cCube* pulls the required Docker images from the repository (usually DockerHub, unless using a private repository), deploys the services, and enqueues the new tasks [1]. The view on *cCube* from an end-user perspective is depicted in Figure 2.1. We specifically included this figure since *PGAcloud* follows the same principle as *cCube* regarding the end-user view.

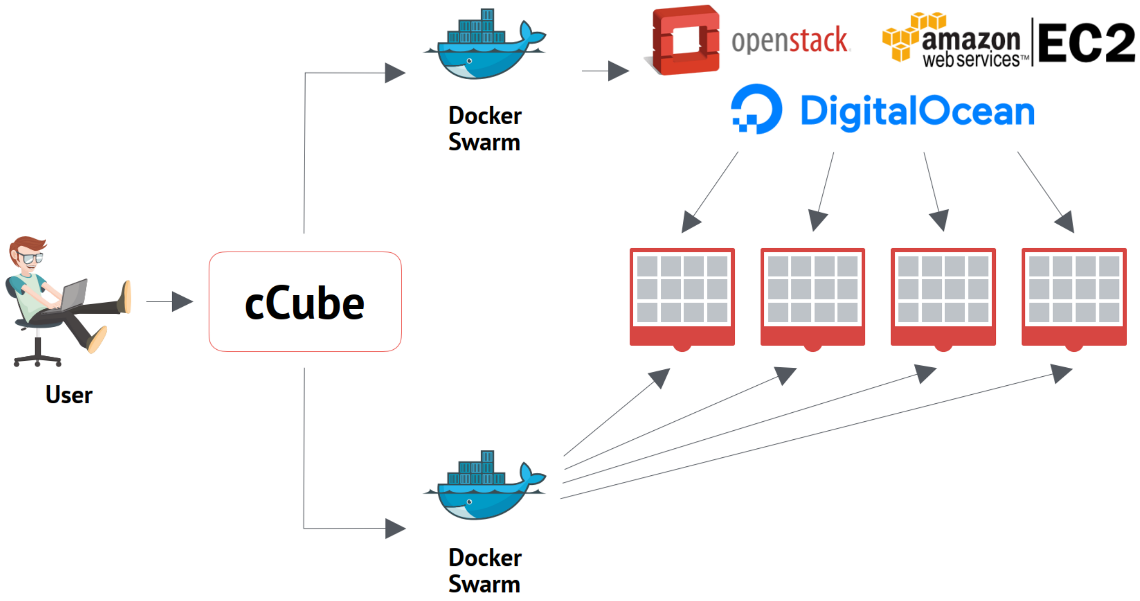

Figure 2.1: *cCube* from the perspective of an end-user. The same view applies to *PGAcloud* end-users. Source: [1].

Chapter 3

Framework Design

We propose *PGAcloud*, a framework for cloud deployment of any local implementations of Genetic Algorithms written in any programming language. We can divide the architecture of our framework into three essential units: the local *Client*, which can be downloaded from the *PGAcloud* repository and serves the user as an interface concerning framework-related interactions; the cloud *Manager*, that receives cloud-related requests and orchestrates the software containers in the cloud environment; multiple runs of Parallel Genetic Algorithms, deployed and computing in parallel – fully isolated from each other.

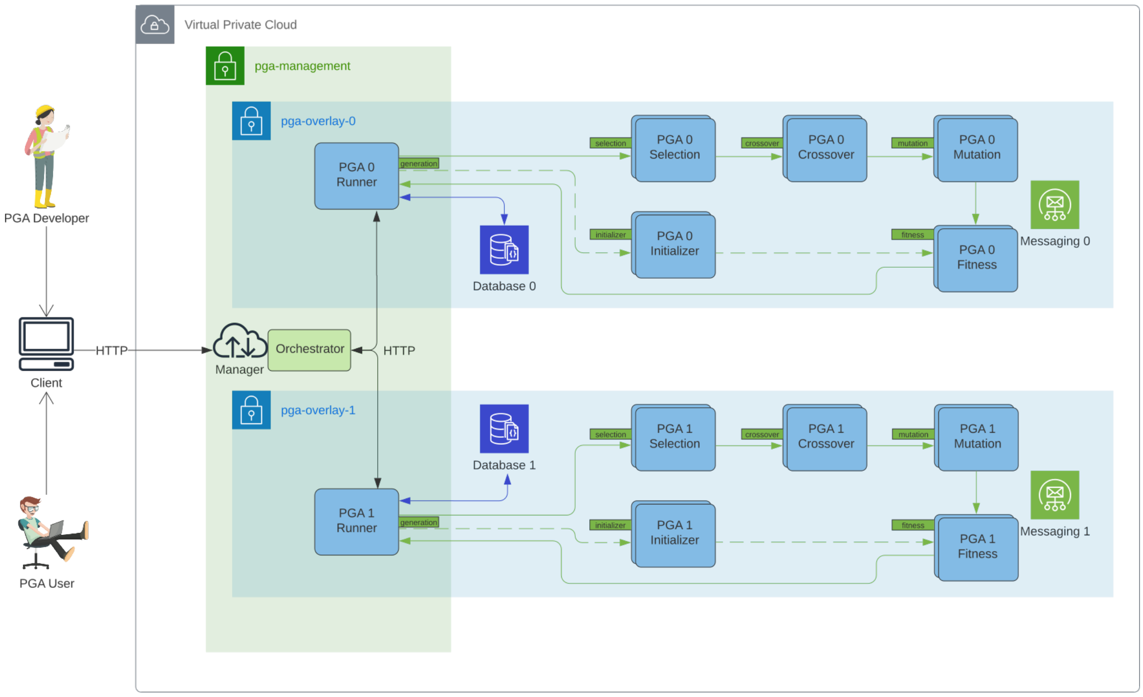

Figure 3.1: The architecture of the *PGAcloud* framework. *(Icons partially from [2])*

Local Client. Interaction with the framework from outside the cloud is possible through the local *Client*, separated from the cloud environment. It provides a command-line interface with commands for any interaction with the remote environment. Such interactions currently include cloud setup and teardown (e.g., for a cloud provider who wants to set up the virtual machines), or PGA related commands like creation, manipulation, and teardown (of specific PGAs).

To create the cloud environment and initialize the *Manager*, the *Client* provides the two commands `client cloud create` and `client cloud init`. When trying to establish a secure connection to the cloud *Manager*, the user must provide a path containing valid SSL certificates. Algorithm 2 on page 29 in the appendix further explicates the generally intended workflow for creating, running, and terminating a PGA computation.

Once the cloud is prepared and the *Manager* running, the user must download the PGA configuration template from the *Client* repository. It includes the container details (e.g., which image to use or how many instances to deploy) and other PGA related properties that can be defined. For improved reproducibility in scientific experiments, the configuration file also includes a field to specify a seed for seeding the random generators used in the different components. Furthermore, suppose a custom operator is implemented. In that case, there is the possibility to declare additional file paths for files that are used by the algorithm and need to be distributed in the cloud. Finally, the path to the configuration file must be provided as an argument to the *Client* command for PGA creation, `client pga create`.

Cloud Management. The only access point within the cloud from the outside is the cloud *Manager*. It provides an API for every allowed interaction and will forward requests where necessary. The *Manager* container also contains the *Orchestrator*, whose implementation is chosen according to the cloud provisioner (e.g., Docker or Google Kubernetes[1]).

The *Manager* receives incoming requests, parses PGA configurations and models, and uses the *Orchestrator* to deploy and regulate affected PGA components according to the model or specific modification requests (i.e., deployment, scaling, removal). To ensure isolation of unique PGA runs, while still being able to communicate with them, the *Manager* is part of the "Management Network."

When creating a new PGA run, the user can provide paths for additional files in the configuration. All received files are stored on the *Manager* for potential future usage and, additionally, distributed to every PGA component. This ensures that each possible customization of supporting services or genetic operators could access the additional files if necessary.

Parallel Genetic Algorithm. Each PGA contains a *Runner* instance, providing an API for PGA related actions and serving as an anchor point accessible to the *Manager*. The *Runner* is the head of each PGA run deployed to the cloud. However, it is not directly accessible from outside the cloud, making the *Manager* the one in charge.

The *Runner* is part of both the "Management Network," allowing two-way communication with the *Manager*, as well as the specific "PGA Network," containing all other PGA components. Creating a separate network for each PGA run ensures the isolation of the components and prevents communication across multiple PGAs. Contained in two networks at once, the *Runner* acts as a mediator between the *Manager* and isolated PGA components like the genetic operators or the database instance.

After instantiation, the *Runner* distributes the properties and handles the creation of an initial population. If the user provided a file containing initial solutions, the *Runner* reads and parses its contents to valid *Individual* objects. The resulting population is then sent directly to the *Fitness Evaluation*. If no population is provided, the *Runner* retrieves the desired population size from the

---

[1] https://kubernetes.io

Algorithm 1 Initializing Population

```
if $initial\_population$ then
    Runner.parse_individuals($initial\_population$)
    Runner.send_evaluation($initial\_population$)
else if not $initial\_population$ then
    Runner.request_individuals()
    $generated \leftarrow$ Initializer.generate_individuals()
    for $individual$ in $generated$ do
        Initializer.send_evaluation($individual$)
    end for
end if
```

configuration and delegates the generation of individuals to the *Initializer*. The decision tree and corresponding actions are depicted in Algorithm 1.

Apart from that, the *Runner* is responsible for the PGA computation: it coordinates the workflow according to the given model and stores the population after each generation. The PGA computation workflow will be split into different genetic operators. There is an algorithm for the "Master-Slave" model in the appendix, see Algorithms 3 and 4 on page 30.

PGAcloud aimed at allowing PGA developers to include custom implementations of genetic operators, similar to what *cCube* [1] can offer. Because any genetic operator could be subject to custom implementation, we designed an omni-purpose *Agent* capable of wrapping *any* external algorithm.

The *Agent* hides the integration into the actual PGA environment from the PGA developer, who only needs to provide the required files and his implementation. He can then build upon the *Agent*'s Docker image to inject the code into its container. To guide the interaction between the *Agent* and the developer's implementation, we composed a dedicated configuration file. This file declares essential parameters like the input and output type (either "value" or "file"), the file paths for input and output files, and – most importantly – the specific command stating how to call the algorithm.

Chapter 4

Prototype

With the current *PGAcloud* implementation, we constructed a prototype offering most of the features we want to see in a final version of the framework. The version we present in this work contains the most critical features and is largely operational. In the upcoming Section 4.1, we list the technologies we used to implement our prototype, explain their core concepts, and elaborate on why we chose to exploit them. We will dive into the specifics of our prototype within the following sections of this chapter (Sections 4.2 and 4.4). Nevertheless, some aspects are still not fully developed and might require refactoring in the future. We point out some of these aspects in a section dedicated to present limitations and known issues (Section 4.3).

4.1 Chosen Technologies

Docker. For orchestrating the *PGAcloud* environment we decided to use *Docker*[1]. The Docker toolbox was one of the first mature products to produce, deploy, and manage software containers. Since its release in 2013, OS virtualization became more popular "due to much higher performance and flexibility in comparison with a traditional, hypervisor-based virtualization, offering sufficient isolation for numerous applications" [18, 19].

It isolates any created container from the underlying operating system: the container builds upon an independent virtual file system and the code of a very basic Linux OS (not the OS itself, however). This code allows the execution of system-related commands, e.g., modification of the file system.

Salza and Ferruci [2] stated the following regarding containerization:

> From the application perspective, there is no difference between an execution on a dedicated machine and inside the container: the application is run in a short time in a full isolated Linux environment and can find others only by using the network. This reduces drastically the activities of installation and maintenance of applications: configuration management methodologies can define the environments and the application can be tested during the process from development to actual production execution, in a CI fashion.

To build a container, Docker requires a blueprint of what to build, the so-called "image." This image contains installation instructions and the necessary code to run the application inside. Creating such an image is possible in two ways: the first one would be to execute the desired operations directly inside a running container and then to save its final state. The other would

---

[1] https://www.docker.com/

be to execute a "Dockerfile," a file containing a set of instructions that conform to a particular syntax. The latter is usually the preferred method because a Dockerfile can be easily maintained with the rest of the source code. After creating it, an image can be uploaded to and later be downloaded from an online registry (either the official *DockerHub*[2] provided by Docker itself, or a private registry) [2].

The main advantage of using images is that they only need to be built once before uploading it to the registry. When used for creating new containers, the installation and build operations will not have to be repeated [2].

Initially, Docker was only able to execute containers on single machines (i.e., with Docker Compose). The older Docker Compose would manage and deploy a containerized application on a single host [20]. However, it did not take long for other cluster orchestrators to appear, e.g., Docker Swarm or Google Kubernetes, which are perfectly able to orchestrate containers in a cloud over multiple machines [18].

PGAcloud uses Docker-Machine to provision the host nodes and Docker Swarm to orchestrate the cloud. Using its built-in load balancing feature, Docker, through Docker Swarm, manages the creation and execution of containers on the available machines in the cloud. "The powerful feature of Docker of executing an entire environment makes possible the implementation of any genetic Operator, in any preferred programming language or using any external tool" [2].

Python. *Python*[3] is a popular and beginner-friendly programming language. It supports object-oriented programming and is great for data processing. We chose to use Python because of its ease-of-use and its large variety of provided tools and libraries.

Docker-Py. One of the many tools Python provides is *docker-py*[4], the official SDK for the Docker Engine API. It is consistently updated and maintained as an open-source project.

docker-py enables Docker-specific actions in Python, which would otherwise only be accessible on the command line. However, *docker-py* also had some limiting factors where its API or underlying implementation was flawed (as any software is in some way). In some cases, it did not allow to perform actions it was supposed to support, or simply could not provide the desired functionality. Examples of this were, on the one hand, the cloud setup with *Docker-Machine*, for which *docker-py* simply does not include the tools required for machine commands. On the other hand, there were issues when creating or updating Docker services with *Docker Secrets* or *Configs*, because these objects are not JSON serializable for some reason (supposedly a bug in the underlying implementation). Additionally, for our framework implementation, we were unable to use built-in health check methods to verify that a scaled Swarm Service has been created and is running and ready to receive API calls. In all mentioned cases, the workaround was to write and execute a bash script, achieving the same results on a more technical level.

Click. *Click*[5] is a Python package providing a composable command-line interface to any user with very little code. It is easily implemented, highly configurable, and has a wide range of valuable features, such as command structures and hierarchy; options and arguments; supporting paths and file existence checks; easy access to user feedback.

Flask. *Flask*[6] is a microframework for Python, developed by the creators of *Click*. Using *Flask*, it is possible to quickly set up a basic but powerful JSON-based REST API that is manageable

---

[2] https://hub.docker.com

[3] https://www.python.org/

[4] https://docker-py.readthedocs.io/

[5] https://click.palletsprojects.com/

[6] https://flask.palletsprojects.com/

without unexpected complications. Because *Flask* avoided including mostly unused extensions, it is lightweight and ideal for deployment in a cloud environment.

RabbitMQ. *RabbitMQ*[7] is an open-source "message broker," which implements the Advanced Message Queueing Protocol (AMQP). In its purest form, RabbitMQ provides a queue able to accept messages from a host A and forward them to host B (for simplification, the actual workings under the hood are abstracted through the use of default exchanges). These messages can contain either plain text or blobs of binary data. Figure 4.1 illustrates the RabbitMQ message flow.

The message exchange with the queues follows the Producer-Consumer-Pattern. "If the publisher and consumers are connected to a queue, they can communicate with each other without actually knowing each other. It makes RabbitMQ a powerful tool for scalable distribution of tasks since it is possible to add and remove participants without breaking the communication" [2].

In addition to this already valuable feature, RabbitMQ also provides a prepared Docker image for easy deployment, e.g., in a containerized cloud environment. Using the message broker as the only form of inter-container communication within a single PGA run, we are free to scale individual operators at runtime using the provided features of Docker Swarm without interrupting the message flow. Considering that for a new container to join the computation at any given time, it must first discover the RabbitMQ service. To ensure this, we used a static DNS name for the service. Technically, the user can configure this name, but it is already included in the configuration template as "rabbitMQ."

All in all, RabbitMQ is a powerful tool for message queues. Although there might be more recent and slimmer substitutions available, RabbitMQ is still popular and easily implemented. The main challenge is understanding its concepts and inner workings. After that, it is all about adequately configuring the exchanges and queues to avoid communication overhead as much as possible.

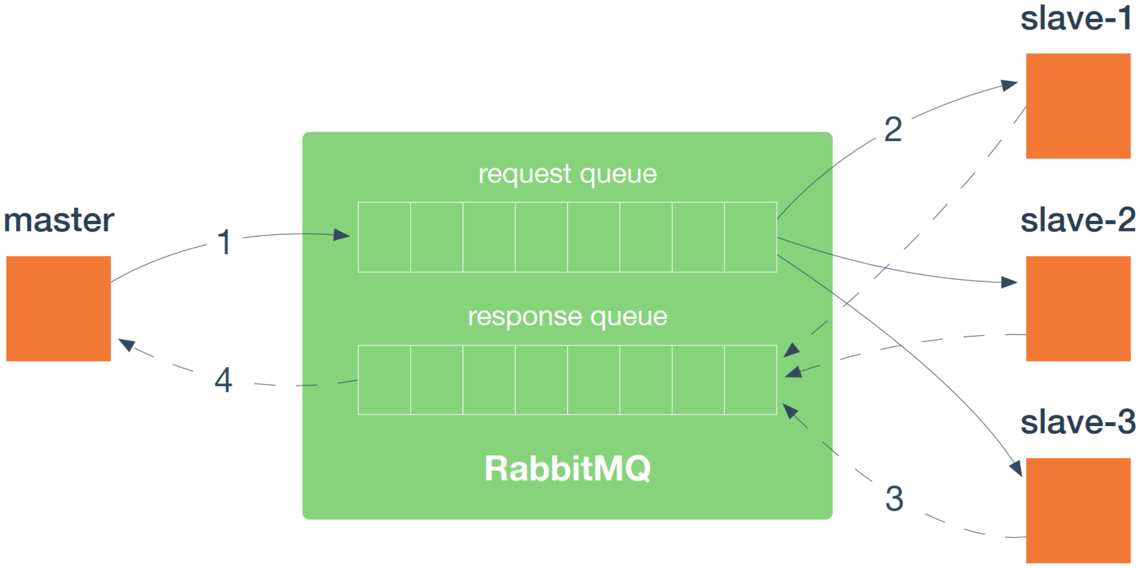

Figure 4.1: The RabbitMQ message flow through queues. Source: [2].

---

[7] https://www.rabbitmq.com/

Redis. *Redis*[8] is an in-memory database storage module that allows storing data structures. A user can perform atomic operations on the stored content. From a developer's perspective, Redis allows storing data in the form of integers, (serialized) strings, blobs of binary data, or lists consisting of such. In combination with a custom de-/serializer, any custom object can be stored in a Redis database.

Additionally, Redis provides an alpine docker image (i.e., an image optimized towards using as little space as possible), one of the most popular container images in 2020 so far [21], and can be included in many cloud environments.

4.2 Framework Components Implementation

At the current state of development, *PGAcloud* only supports the "Master-Slave" model for parallelization, although genetic operators other than the *Fitness Evaluation* can also be scaled. Customization is possible with the wrapping *Agent* container and direct code access to the *Initializer*.

To demonstrate the usage of the *PGAcloud* framework, we included an example implementation based on the Knapsack problem (described in Section 4.4). We provided customized configuration files with additional properties required by our custom fitness evaluation function. As a proof of concept, this fitness evaluation is written in Java and highlights the ability to include external code written in any language with our proposed *Agent*.

Usage-wise, the *PGAcloud* framework is mostly closed source. Adapting it to solve different problems requires changes to the user-provided configuration files and the *Initializer* component to customize the *Individual* model for generating an initial population according to the specific problem to solve. Other genetic operators should, in theory, not have to be adapted. However, they are restricted in terms of the *Individual* model, which demands possible solutions be of type string and the fitness value of type float. Also, the naming of the fields is predefined. If an external operator made use of fields other than `solutions` or `fitness`, they would not be passed on by default without adjustments to the *Individual* model.

Client. The local *Client* uses Docker-Machine to create and configure the cloud environment for docker orchestration with Docker Swarm. Since Docker-Machine is not part of the regular Docker distribution, it must be installed manually by the user in advance alongside Docker Desktop.

We decided to use Click to provide a simple command-line interface to the user. Unfortunately, each Click command represents a new entry or session in the *Client* interaction. So, we must somehow pass relevant information over multiple sessions. We solved this issue by storing essential and non-retrievable information in a local context file in the YAML format. We allow explicitly changing these context configurations with the following commands:

- `client config master_host`: Update the configuration for the master host (IP address or hostname) in the cloud. If no argument is provided, it will print the current configuration.

- `client config master_port`: Update the configuration for the exposed port on the master host to map to containers. If no argument is provided, it will print the current configuration.

- `client config certificates`: Update the configuration for the path to the SSL certificates required for secure connection to the cloud master host. At the given location, the files "ca.pem," "cert.pem," and "key.pem" must be found to establish a valid SSL connection. If no argument is provided, it will print the current configuration.

---

[8] https://redis.io/

As has been previously established, the *Client* and the *Manager* communicate over a secure HTTPS connection, encrypted with SSL. The user needs to provide valid SSL certificates for this connection, which we pass on to each cloud component as Docker *Secrets*. Those *Secrets* are encrypted by Docker and not accessible as clear text.

Regarding the chosen genetic operators, the *Elitism* operator is built-in and thus not optional. It takes a certain percentage as property from the configuration file and retains this percentage of best individuals from the population for direct integration at the end of each generation (when *Survival Selection* occurs). Theoretically, the *Elitism* operator could be extracted as a separate container in the PGA model. The same goes for the *Survival Selection*.

For regular cloud or PGA related interactions, the *Client* provides the following commands:

- `client cloud create`: Create and set up the cloud environment.
- `client cloud init`: Initialize the PGA Manager.
- `client cloud reset`: Reset the cloud by removing the PGA Manager.
- `client cloud destroy`: Remove the cloud environment and all its PGA contents.
- `client pga create`: Create a new PGA run.
- `client pga start`: Start computation of given PGA.
- `client pga stop`: Stop computation of given PGA and remove it.
- `client pga monitor`: Monitor computation statistics of given PGA (currently fittest individual, generation, computation time, Etc.)
- `client pga pause`: Pause given PGA after finishing the current generation.

Manager. The *Manager* represents the interface between the cloud environment and the local *Client*. It provides a Flask API for communication with the *Client* and each PGA *Runner*. Its primary responsibilities include receiving requests for PGA interaction and processing or forwarding its contents. The *Manager* container also includes an abstract *Orchestrator* class, which defines the desired behavior for container orchestration (i.e., creation, scaling, modification, removal) each implementation of an *Orchestrator* must provide. Following the concept of "separation of concerns," it is irrelevant to the *Manager* how exactly the *Orchestrator* achieves this.

When creating a new PGA run, the user can provide paths for additional files in the configuration. The Docker *Orchestrator* converts each file into a Docker *Config* suited to the newly created PGA. To ensure each possible customization of supporting services or genetic operators has access to these files, the Docker *Orchestrator* attaches the Docker *Configs* to every container related to said PGA run.

We decided to parse the PGA model for new PGA runs in the *Manager* to facilitate "dumb" containers in the "PGA Network." Each container will receive a supplementary, tailored container configuration file containing the PGA ID for identification and exploration purposes, as well as the messaging source (its own queue) and target (where to send processed individuals). The only container with a little more content is the *Runner*, which distinguishes different flows based on the PGA model or configurations. With this information, the *Manager* delegates the creation of required containers to the *Orchestrator*.

Newer versions of Docker offer configuration of different types of Docker *Networks*, i.e., bridge, host, macvlan, or overlay. By introducing the dedicated PGA overlay networks, we isolate the PGAs from each other as much as possible. This isolation allows scaling with multiple PGA runs in parallel as well as proper handling and management. Like so, the specific containers can be isolated from the *Manager* and other PGAs, since the *Manager* only needs to communicate with a PGA's *Runner*. That is the reason we chose to bundle the *Manager* and each created *Runner* within the same "Management Network," where each *Runner* then serves as an intermediary to a specific "PGA Network."

The different stages of the workflow are implemented by the *Initializer* and the genetic operators (i.e., *Selection*, *Crossover*, *Mutation*, *Fitness Evaluation*). They are separately deployed or destroyed by the *Orchestrator* and can be scaled at runtime. The different components communicate by message queues. To ensure the stages are correctly arranged, each operator retrieves the name of the message queue it is supposed to listen to from a customized configuration file, prepared by the *Manager* when parsing the model. After processing any incoming message, the operator reads the name of the target queue from the config file and forwards it accordingly. Transmission of the *Individual* objects by message queues takes place using the JSON data-interchange format.

Runner. The *Runner* is the head of each PGA run deployed to the cloud and coordinates the workflow of the PGA computation. Throughout the PGA workflow communication, we decided to send only single *Individual*s. Although this produces a higher quantity of messages, the network load is probably not significantly higher than when combining multiple *Individual*s in a single message (the data is the same, additional headers make the difference). The only exception to this is the *Selection* operator, which requires the entire population.

Providing an API to the *Manager* allows future additions, like implementing dynamically adjustable properties for an entire PGA or a single operator. However, this is only a possible extension and has not been implemented yet.

Initializer. The *Initializer* listens to a specific message queue for generation requests, processes it, and sends the generated individuals to the *Fitness Evaluation* for an initial evaluation. It can be scaled to reduce the time required for generating a sufficiently large population.

To reduce the loss of work in the case of a failing *Initializer* container, we decided to request the generation of *Individual* objects by sending a single message for every individual. Changing this back to request multiple *Individual*s at once would only require adjusting the messages sent since the corresponding code on the *Initializer* is already in place.

Every request for generation includes *Individual* IDs to differentiate the random values used in the generation process when generating at the same time (usually, random generators are based on the current timestamp, possibly resulting in the same values when running in parallel). One possible way to do so would be to add the ID to the random value, such that even when two random values were identical, they could be differentiated.

Fitness Evaluation. The *Fitness Evaluation* listens to its queue for individuals to be evaluated. It reads the `solution` property, compares it to the specific problem at hand by consulting relevant properties (in the case of the Knapsack problem, this would include the knapsack capacity or the weights and profits of all available items), and converts its fitness to a numeric value. After evaluation, each individual is sent back to the *Runner*.

In the Knapsack demonstration, the *Fitness Evaluation* was implemented in Java. This implementation intended to prove that it is possible to provide the code for an external operator written in a different language and successfully integrate it into the PGA workflow. Since it was not relevant for the proof of concept, our implementation is purely functional and in no way optimized towards quality or performance.

Selection. Once the *Runner* received back all individuals, it releases them to the PGA model (Algorithm 3 in the appendix). In the case of the "Master-Slave" model, the first destination is the *Selection* operator. Here, the entire population is split into pairs later used for mating.

In the current version of the *PGAcloud* framework, only the "Roulette Wheel Selection" has been implemented. However, the possibility is already provided to expand the list of available selectors inside the container by implementing others, like the "Tournament Selection" or "Rank Selection" [22], for example. Alternatively, after some adjustments, the entire container could be replaced by a custom implementation wrapped by the *Agent*.

Crossover. The *Crossover* operator receives the pairs from the *Selection*. It crosses two individuals according to the defined crossover rate, possibly producing two new offspring individuals. *Crossover* then sends the final two individuals to the *Mutation* operator.

In the current version of the *PGAcloud* framework, only the "One-Point Crossover" has been implemented. Again, it is possible to add other crossers like the "Multi-Point Crossover" or "Uniform Crossover" [23] to the currently available crossers. Alternatively, after some adjustments, a custom implementation could replace the entire container.

Mutation. The *Mutation* operator processes any request for mutation of an individual. Like the *Crossover* operator, it also retrieves the predefined mutation rate from the configuration file. If the random generator matches the mutation rate, the individual is mutated. In the end, any individual is directed to *Fitness Evaluation*.

In the current version of the *PGAcloud* framework, only the "Bit Flip Mutation" has been implemented. Of course, the available mutators can be supplemented by other mutators, e.g., the "Inversion Mutation," "Scramble Mutation," or "Swap Mutation" [24]. Alternatively, after some adjustments, the entire container could be replaced by using the *Agent* to wrap a custom implementation.

Agent. The *Agent* is the only relevant component of our framework for integrating the algorithm for a custom operator. It is required to tap into the underlying message flow and maintain the workflow chain such that the external algorithm does not have to be bothered with that. The *Agent* ensures retrieval and processing of any individual directed to the included operator and forwarding the computation results according to the defined PGA model.

After retrieving the individual from the message queue, the *Agent* will create a file in the input file path if required and write the serialized *Individual* to it. It then calls the operator code with the command parameter (from the *Agent* configuration) on the command line. The external algorithm's responsibility is to write the resulting *Individual* to the file in the output file path. After completion, the *Agent* can read the file, parse the *Individual* inside and send it to the next destination in the workflow. Both files are removed after processing the individual, to ensure a fresh start for the next request.

For more information on how to integrate an external algorithm into the *Agent* container, please consult Section 4.4 about our example Knapsack implementation.

When including a custom implementation into the *Agent* container, using Docker's relatively new "multistage builds"[9] results in a much slimmer Dockerfile. However, one should keep in mind that each new layer separates any underlying container layers.

This separation is especially important if multiple different programming languages are requested: when trying to build from an OpenJDK image and then basing on the *Agent*'s Python image, the Java installation is no longer accessible to the *Agent*. In addition, defining a custom entry point with the "CMD" statement would overwrite the *Agent*'s underlying entry point, effectively preventing it from executing its message queue script and listening to the assigned queue.

---

[9] https://docs.docker.com/develop/develop-images/multistage-build/

4.3 Open Refinements and Limitations

The *Agent* container can not be used to implement the *Selection* operator because the *Agent* is designed to receive exactly one *Individual* object. The *Selection* operator, however, must receive an array for the entire population.

As one of the last features, we provided a random seed configuration to improve reproducibility in scientific experiments (mentioned in Chapter 3). Unfortunately, this seed is not in use at the current stage of development. However, it is intended for usage in any cloud component (i.e., the *Initializer* and any genetic operator making use of random values).

Regarding the general aspect of software engineering, the *PGAcloud* framework is dependent on being served with valid configurations. It is error-prone and unable to correctly handle faulty configuration files or misuse by the user (either intentional or due to lack of understanding). Nonetheless, if encountering an error is desirable, it should be handled accordingly, and proper feedback to the user should be ensured. In general, making the code more robust is of interest to any PGA developer. Moreover, the framework currently lacks an extensive documentation on how to (properly) use the *Client* and all the possibilities of the configuration files.

As a final point, the previously mentioned error propagation from internal PGA components upstream to the user should be refined. At this time, it is not possible to propagate any error messages or tracebacks. In the event of a runtime exception, the user has no information on what exactly went wrong (unless directly accessing the server logs is possible). This uncertainty is due to the original HTTP request – which originated from the *Client* and was sent to the *Manager* – being forwarded to the *Runner*. The PGA workflow it manages is entirely reliant on message queues. Suppose any genetic operator encounters an issue while processing the individuals. In that case, the *Runner* does not know about it and never will, preventing it from returning feedback to the *Manager*, and lastly, the *Client* or user, respectively.

4.4 Example of Use

To demonstrate that our proof of concept works as intended, we provided an example implementation of the *PGAcloud* framework integrating a customized *Fitness Evaluation* operator to solve the Knapsack problem[10].

Knapsack Problem. The Knapsack problem is a well-known optimization problem that is simple enough to be used to evaluate the functionality of our framework. This problem revolves around a hypothetical bag of limited capacity (the "Knapsack") and a set of predefined items, each having a certain weight and value. The goal of the problem is to achieve the highest profit by filling the Knapsack with items, i.e., choosing some items and throwing away the rest of them. This selection makes the Knapsack problem a multi-objective problem, optimizing the weight and value of the selected items.

Implementation. In our example implementation, we included some custom properties in the PGA configuration file to accommodate it to the problem at hand. We added a knapsack capacity, a list of weights, and a list of profits (values). The combination of both lists represents the list of items available for choosing. Since the PGA configuration properties are stored in the Redis database, we can retrieve any required properties in every container.

To solve our Knapsack problem, we did not have to adjust any other genetic operators, so they were hardcoded in their functioning. The "Master-Slave" model instructs the operators to send

---

[10] $https://en.wikipedia.org/wiki/Knapsack\_problem$

individuals from the *Runner* to *Selection*, *Crossover*, *Mutation*, and, finally, to the *Fitness Evaluation*. The last-mentioned is the only operator that was implemented with the *Agent*.

The full configuration file can be found in the appendix, see Listing 5.1 on page 31. Note that the *Agent*'s image is not part of the provided PGA operators.

```
model: "Master-Slave"
operators: {
   SEL: {
      name: 'selection', image: 'jluech/pga-cloud-selection',
      scaling: 2, messaging: "selection"
   },
   CO: {
      name: 'crossover', image: 'jluech/pga-cloud-crossover',
      scaling: 2, messaging: "crossover"
   },
   MUT: {
      name: 'mutation', image: 'jluech/pga-cloud-mutation',
      scaling: 2, messaging: "mutation"
   },
   FE: {
      name: 'fitness', image: 'jluech/pga-cloud-fitness',
      scaling: 1, messaging: "fitness"
   },
}
properties: {
   knapsack_capacity: 150,
   item_count: 30,
   items_weights: [10, 22, 33, 17, 7, 14, 16, 24, 8, 12, 30, 21, 13, 44,
      25, 6, 27, 18, 9, 40, 11, 28, 3, 14, 35, 26, 50, 18, 29, 10],
   items_profits: [21, 35, 25, 17, 11, 21, 8, 10, 13, 18, 29, 23, 9, 33,
      54, 22, 16, 31, 20, 37, 24, 28, 7, 19, 28, 3, 43, 26, 12, 14],
}
```

Listing 4.1: Knapsack PGA Configuration

That is due to the Docker image of our fitness function basing on the *Agent*'s image, extending and ultimately replacing it.

The full Dockerfile can be found in the appendix, see Listing 5.2 on page 33

```
// Copy contents to container.
COPY . /operator
WORKDIR /operator
// Start with PGAcloud Agent as base image.
FROM jluech/pga-cloud-agent:latest
// Insert agent container configuration file.
COPY --from=0 /operator/custom-config.yml /pga/custom-config.yml
// Create jar file from custom operator in PGA agent.
COPY --from=0 /operator/main /pga/main
RUN javac /pga/main/fitness/ *.java && cd /pga/main && \
   jar cfe FitnessEvaluation.jar fitness.Main fitness
```

Listing 4.2: Fitness Evaluation Dockerfile

To properly include our algorithm, we provided the following *Agent* configuration. The full configuration is included in the appendix, see Listing 5.3 on page 34.

```
container_name: "fitness"

property_keys:
- {key: 'knapsack_capacity', is_list: False}
- {key: 'items_weights', is_list: True}
- {key: 'items_profits', is_list: True}

input_type: "file"
output_type: "file"
input_path: "/pga/input.yml"
output_path: "/pga/output.yml"

command: "java -jar main/FitnessEvaluation.jar
   capacity={knapsack_capacity} weights={items_weights}
   profits={items_profits}"
```

Listing 4.3: Agent Configuration

With the provided command string, we request to be passed the values of our Knapsack properties. The Agent reads the property keys inside the command, retrieves the corresponding values from the Redis database, and replaces the keys in the command with the actual values. For the items' weights and profits, the list items are passed as a comma-separated string.

By setting the input/output type to "file," we instruct the *Agent* to write the individual it receives from the message queue as a serialized dictionary to the given input file. Since we copied our code into the *Agent*'s container, we have access to its internal file system. We can then read the individual from the input file, parse it to an *Individual* object, and compute its fitness value.

```
double fitEval(Individual individual) {
   double profits = 0;
   double used_weight = 0;
   double available_weight = 0;
   for (int i=0; i < individual.solution.length(); i++) {
      int bit = Integer.parseInt(individual.solution.substring(i, i+1));
      profits += (bit * this.items_profits.get(i));
      used_weight += (bit * this.items_weights.get(i));
      available_weight += this.items_weights.get(i);
   }
   double scaled_unused_weight = available_weight
      * Math.abs(capacity - used_weight);
   double fitness = profits - scaled_unused_weight;
   individual.fitness = fitness;
   return fitness;
}
```

Listing 4.4: Fitness Function

The full implementation of the fitness function is located in the appendix, see Listing 5.4 on page 35. In our fitness function, we go through the binary solution string of the individual. Each bit defines if the item at position *i* (represented by the weight and value at position *i* of our list properties) will be included in the Knapsack: "1" means include, "0" means ignore (the added profit or weight is zero). We then count the profits we achieve with this solution and the weight of both the chosen as well as all available items. Finally, we scale the unused weight to punish deviations from the full Knapsack capacity (also accounting for overcapacity). The fitness value results from the achieved profits minus the scaled unused weight.

```java
public static void main(String[] args) {
   // Collect individual from input file.
   Map<String, String> ind_map = input_data.get(0);
   Individual individual = new Individual(ind_map.get("solution"),
      Double.parseDouble(ind_map.get("fitness")));

   // Parse input data and create fitness operator with it.
   FitnessEvaluation fitEval = new FitnessEvaluation(profits, weights,
      Double.parseDouble(argVals.get("capacity")));

   // Evaluate the fitness of the given individual.
   fitEval.fitEval(individual);

   // Write individual with computed fitness to output file.
   List<String> content = Arrays.asList(
      "{",
      String.format("\t\"solution\": \"%s\",", individual.solution),
      String.format("\t\"fitness\": %f", individual.fitness),
      "}"
   );
   Path file = Paths.get(output_path);
   Files.write(file, content, StandardCharsets.UTF_8);
}
```

Listing 4.5: Fitness Evaluation Operator

Next, we update the *Individual*'s fitness value attribute and write it back to the output file – again as a serialized dictionary. After terminating our algorithm, the *Agent* resumes its routine, reads and parses the evaluated individual from the output file, and sends it back to the *Runner*. With this, we completed the generation cycle of the "Master-Slave" PGA model.

The code for the `main` method of the Fitness Evaluation operator that is displayed here has been shortened considerably. The full method can be viewed in the appendix (refer to Listing 5.5 on page 36).

Chapter 5

Conclusions

We proposed *PGAcloud*, a framework capable of including custom implementations of Genetic Algorithms and deploying them to a prepared cloud environment. It addresses two kinds of users: PGA developers and PGA end-users.

The PGA developers can access the relevant code from the *PGAcloud* repositories and include their custom implementation – along with the necessary configurations or additional files – into the *Agent* container, effectively deploying a local algorithm to the cloud. This algorithm does not require any cloud capabilities or knowledge about the internal workings of the *PGAcloud* framework. After including the custom operator, the PGA developer's role changes to the role of a PGA end-user.

A PGA end-user can interact with a finished instance of the *PGAcloud* framework through the local *Client*. It provides templates for configuration files containing the PGA model, specification of images to deploy the cloud components, and PGA-related properties. After any adjustments, the user simply needs to execute the desired commands from the *Client*'s command-line interface and point it to the desired PGA configuration file. Once the computation has started, *PGAcloud* will take care of distributing each component and balancing the load on the cloud hosts. When finished, the fittest individual will be reported back to the user on the command line output.

The proposed framework is currently restricting in several ways: firstly, the user must provide valid configurations and strictly adhere to the guidelines stated in the templates. Secondly, a PGA developer must download and adapt the *Initializer* repository (i.e., any model of an *Individual*) when trying to implement a PGA for a different problem than the Knapsack problem. Lastly, with the previous restriction in mind, the model for an *Individual* relies on specifically named fields of a particular type, i.e., `solution` of type string and `fitness` of type float. Adding and transmitting any other properties is not provided by default and may require changes to the code of any genetic operator or the *Agent*.

With our proposition, we aimed to implement a proof of concept framework that can include any Genetic Algorithm and deploy it to the cloud in a flexible and easily scalable way. Since this goal is purely functional, we did not provide any notion of performance and consider corresponding measurements to be out of scope (at least for now).

Future Work. The current state of the *PGAcloud* implementation offers many approaches for improvements. It would be useful to extend it to support other cloud providers (e.g., AWS, OpenStack, Virtualbox, Etc.) or cloud orchestrators (e.g., Google Kubernetes; currently, only Docker is operational).

Also, as there are different levels of parallelization of a GA, it would make sense to extend the framework for other PGA models like the "Island" or "Grid" model. Besides, when implementing the "Island" model, there are several approaches to dynamically manage the number of islands, as have been proposed by Dziurzanski et al. [18].

Considering that we already allow custom operators and provide the configuration for custom models, it should also be possible to configure and deploy them.

With our example implementation, *PGAcloud* is limiting the application of the *Agent* to the *Fitness Evaluation* operator. All other operators are hardcoded in their behavior and not customizable. Changing this for full customization of every operator is easily possible but would require some investments to be completed. These adjustments would concurrently allow offering different implementations of each operator, as has already been mentioned in the description of the resulting software products.

The original scope of this work was also to include *Client* commands for monitoring, manipulating, or suspending/resuming an active PGA run. Adding these features would increase the flexibility aspect of our proposed framework.

Concerning runtime manipulation of PGA runs: theoretically, using the advantages of providing an API and custom configuration files would not only allow a user to declare initial properties for an entire PGA or a single operator but also enable changing them at runtime. Regarding this matter, it might be useful to consult the survey of Karafotias, Hogendoorn, and Eiben [25] on the trends and challenges of parameter control in Evolutionary Algorithms. Their work is about tuning parameters during execution, which would potentially be possible with our approach.

Bibliography

[1] P. Salza, E. Hemberg, F. Ferrucci, and U.-M. O'Reilly, *Towards Evolutionary Machine Learning Comparison, Competition, and Collaboration with a Multi-Cloud Platform*. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1263–1270.

[2] P. Salza and F. Ferrucci, "Speed up genetic algorithms in the cloud using software containers," *Future Generation Computer Systems*, vol. 92, pp. 276–289, 2019.

[3] P. Salza, F. Ferrucci, and F. Sarro, "Elephant56: Design and implementation of a parallel genetic algorithms framework on hadoop mapreduce," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '16 Companion. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1315–1322.

[4] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.

[5] G. Luque and E. Alba, *Parallel Genetic Algorithms: Theory and Real World Applications*, ser. Studies in Computational Intelligence. Springer Berlin Heidelberg, 2011.

[6] L. Zheng, Y. Lu, M. Guo, S. Guo, and C.-Z. Xu, "Architecture-based design and optimization of genetic algorithms on multi-and many-core systems," *Future Generation Computer Systems*, vol. 38, pp. 75–91, 2014.

[7] S. Yoo, M. Harman, and S. Ur, "Gpgpu test suite minimisation: search based software engineering performance improvement using graphics cards," *Empirical Software Engineering*, vol. 18, no. 3, pp. 550–593, 2013.

[8] M. Ivanovic, V. Simic, B. Stojanovic, A. Kaplarevic-Malisic, and B. Marovic, "Elastic grid resource provisioning with wobingo: A parallel framework for genetic algorithm based optimization," *Future Generation Computer Systems*, vol. 42, pp. 44–54, 2015.

[9] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, and B.-S. Lee, "Efficient hierarchical parallel genetic algorithms using grid computing," *Future Generation Computer Systems*, vol. 23, no. 4, pp. 658–670, 2007.

[10] P. Salza, F. Ferrucci, and F. Sarro, "Develop, deploy and execute parallel genetic algorithms in the cloud," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '16 Companion. New York, NY, USA: Association for Computing Machinery, 2016, pp. 121–122.

[11] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transactions on Cloud Computing*, vol. 5, no. 2, pp. 358–371, 2017.

[12] A. Grajdeanu, "Parallel models for evolutionary algorithms," *ECLab, George Mason University*, vol. 38, 2003.

[13] E. Cantu-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.

[14] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 785–793.

[15] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro, *Towards migrating genetic algorithms for test data generation to the cloud*, 2012.

[16] M. García-Valdez, L. Trujillo, J.-J. Merelo, F. F. De Vega, and G. Olague, "The evospace model for pool-based evolutionary algorithms," *Journal of Grid Computing*, vol. 13, no. 3, pp. 329–349, 2015.

[17] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, 2017.

[18] P. Dziurzanski, S. Zhao, M. Przewozniczek, M. Komarnicki, and L. S. Indrusiak, "Scalable distributed evolutionary algorithm orchestration using docker containers," *Journal of Computational Science*, vol. 40, p. 101069, 2020.

[19] M. T. Chung, N. Quang-Hung, M. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016, pp. 52–57.

[20] R. Singh, "Docker compose vs docker swarm," 2019. [Online]. Available: https://linuxhint.com/docker_compose_vs_docker_swarm/

[21] J. Kreisa, "Docker index: Dramatic growth in docker usage affirms the continued rising power of developers," 2020. [Online]. Available: https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/

[22] Tutorials Point India Ltd., "Genetic algorithms - parent selection," 2020. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm

[23] ——, "Genetic algorithms - crossover," 2020. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

[24] ——, "Genetic algorithms - mutation," 2020. [Online]. Available: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm

[25] G. Karafotias, M. Hoogendoorn, and A. E. Eiben, "Parameter control in evolutionary algorithms: Trends and challenges," *IEEE Transactions on Evolutionary Computation*, vol. 19, no. 2, pp. 167–187, 2015.

Appendix

Pseudocode Algorithms

Intended workflow for creating, running, and terminating a PGA computation:

Algorithm 2 User-Client Interactions

Require: configurations, certificates

Ensure: fittest individual

```
Client.create_cloud($cloud\_config\_path$)
Client.init_cloud($certificates\_path$)

$pga\_id \leftarrow$ Client.create_pga($pga\_config\_path?$) {
  Manager.create_pga() {
    Orchestrator.setup_pga()
    Runner.distribute_properties()
    Runner.initialize_population()
  }
}
$fittest\_individual \leftarrow$ Client.start_pga($pga\_id$) {
  Manager.start_pga($pga\_id$) {
    Runner.start_pga()
    Runner.stop_pga()
  }
}

Client.stop_pga($pga\_id$) {
  Manager.stop_pga($pga\_id$) {
    Runner.abort_pga()
    Runner.stop_pga()
  }
  Orchestrator.remove_pga($pga\_id$)
}

Client.reset_cloud()
Client.destroy_cloud()
```

Starting the PGA computation:

Algorithm 3 Start PGA: Master-Slave Model

```
Ensure: population
population ← receive_individuals()
while not terminated do
    store_population(population)
    check_abort()
    elite ← apply_elitism(population)
    release_to_model(population)

    couples ← Selection.select_parents(population)
    Selection.send_crossover(couples)
    for couple in couples do
        offspring ← Crossover.cross(couple)
        Crossover.send_mutation(offspring)
    end for
    for individual in offspring do
        mutated ← Mutation.mutate(individual)
        Mutation.send_evaluation(mutated)
    end for
    for individual in mutated do
        evaluated ← Fitness.evaluate(individual)
        Fitness.send_runner(evaluated)
    end for

    population ← receive_individuals()
    combine_elite(population, elite)
    survival_selection(population)
    check_termination()
end while
return population
```

Stopping the PGA computation:

Algorithm 4 Stop PGA

```
Require: population
Ensure: fittest
store_population(population)
fittest ← retrieve_fittest(population)
return fittest
```

Complete Listings

Full configuration file passed to the *Client* to solve the Knapsack problem.

```
// Predefined models: "Master-Slave" and "Island" (not ready yet).
// Declare custom models representing one generation cycle.
// Use "/" as separator and first/last image being the runner,
// e.g., "RUN/SEL/CO/MUT/FE/RUN"
model: "Master-Slave"

// List of inter-component services to use in the PGA,
// including their initial scaling.
// Those in uppercase are predefined default services.
services: {
   MSG: {name: ’rabbitMQ’, image: ’rabbitmq:3.8-alpine’, scaling: ’’},
   DB: {name: ’redis’, image: ’redis:6.0-alpine’, scaling: ’’},
}

// List of setup components, including their fixed scaling.
// Those in uppercase are predefined default setup components.
setups: {
   RUN: {
      name: ’runner’, image: ’jluech/pga-cloud-runner’,
      scaling: 1, messaging: "generation"
   },
   INIT: {
      name: ’initializer’, image: ’jluech/pga-cloud-initializer’,
      scaling: 2, messaging: "initializer"
   },
}

// List all images of operators involved, including their initial
// scaling.
// Those in uppercase are predefined default operators.
operators: {
   SEL: {
      name: ’selection’, image: ’jluech/pga-cloud-selection’,
      scaling: 2, messaging: "selection"
   },
   CO: {
      name: ’crossover’, image: ’jluech/pga-cloud-crossover’,
      scaling: 2, messaging: "crossover"
   },
   MUT: {
      name: ’mutation’, image: ’jluech/pga-cloud-mutation’,
      scaling: 2, messaging: "mutation"
   },
   FE: {
      name: ’fitness’, image: ’jluech/pga-cloud-fitness’,
      scaling: 1, messaging: "fitness"
   },
}

// Settings for providing an initial population.
// If using "\" in the file path you should escape them with an
// additional one, like so "\\"
population: {
   use_initial_population: False,
   population_file_path: ’’
}

// Properties for PGA execution.
// Define at least the predefined ones (uppercase), but can also include
// custom properties for custom operators.
properties: {
   USE_INIT: False,
   MAX_GENERATIONS: 1500,
   MAX_UNIMPROVED_GENERATIONS: 300,
   MAX_TIME_SECONDS: 600,
   POPULATION_SIZE: 200,
   RANDOM_SEED: "",
   MUTATION_RATE: 0.3,
   CROSSOVER_RATE: 0.5,
   ELITISM_RATE: 0.05,
   knapsack_capacity: 150,
   item_count: 30,
   items_weights: [10, 22, 33, 17, 7, 14, 16, 24, 8, 12, 30, 21, 13, 44,
      25, 6, 27, 18, 9, 40, 11, 28, 3, 14, 35, 26, 50, 18, 29, 10],
   items_profits: [21, 35, 25, 17, 11, 21, 8, 10, 13, 18, 29, 23, 9, 33,
      54, 22, 16, 31, 20, 37, 24, 28, 7, 19, 28, 3, 43, 26, 12, 14],
}

// Define here any custom file paths. Use the following scheme
// <unique_key>: ’<file_path>’
// Do NOT use the key "population" for it will be used for the initial
// population file defined above.
// If using "\" in the file paths you should escape them with an
// additional one, like so "\\"
custom_files: {
   demo: ’C:\dev\PGAcloud\demo.yml’,
}
```

Listing 5.1: Complete Knapsack PGA Configuration

The Dockerfile of our custom *Fitness Evaluation*.

```
// Define custom operator image.
FROM openjdk:14-alpine
MAINTAINER "Janik Luechinger janik.luechinger@uzh.ch"

// Copy contents to container.
COPY . /operator
WORKDIR /operator

// Start with PGAcloud Agent as base image.
FROM jluech/pga-cloud-agent:latest
MAINTAINER "Janik Luechinger janik.luechinger@uzh.ch"

// Install java in python image.
RUN apt-get update && \
   apt-get -y upgrade && \
   echo oracle-java14-installer shared/accepted-oracle-license-v1-2
      select true /usr/bin/debconf-set-selections && \
   echo "deb http://ppa.launchpad.net/linuxuprising/java/ubuntu focal
      main" tee /etc/apt/sources.list.d/linuxuprising-java.list && \
   apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
      73C3DB2A && \
   apt update && \
   apt-get -y upgrade && \
   apt install -y oracle-java14-installer

RUN java --version

// Insert agent container configuration file.
COPY --from=0 /operator/custom-config.yml /pga/custom-config.yml

// Create jar file from custom operator in PGA agent.
COPY --from=0 /operator/main /pga/main
RUN javac /pga/main/fitness/*.java && \
   cd /pga/main && \
   jar cfe FitnessEvaluation.jar fitness.Main fitness
```

Listing 5.2: Complete Fitness Dockerfile

Presented below is the full configuration of the *Agent* container to include our implementation of the *Fitness Evaluation*.

```
// Copy this file into "/pga/custom-config.yml" once inside
// the container.

// Provide the name of the operator to the agent for config retrieval.
// Use the corresponding name of the PGA config given to the client cli.
container_name: "fitness"

// Declare keys of properties to look for inside the database (properties
// you require for calling your code).
// Remove the brackets and list them like so
// (leave brackets for an empty list):
// property_keys:
// - {key: '<prop_key>', is_list: True|False}
property_keys:
- {key: 'knapsack_capacity', is_list: False}
- {key: 'items_weights', is_list: True}
- {key: 'items_profits', is_list: True}

// Declare how to receive the individual and how to provide the output of
// your computation.
// Choose: input type ["file" || "value"]
// and output type ["file" || "console"].
// If "file" is chosen, provide a file path to the corresponding file
// (not required for other types).
input_type: "file"
output_type: "file"
input_path: "/pga/input.yml"
output_path: "/pga/output.yml"

// State how to call your code, including potential parameters.
// If you require additional properties, declare them using the related
// key used to retrieve it from the DB.
// Contents of lists will be passed as a comma-separated string, without
// whitespaces between consecutive elements.
// "... param={PROP_KEY}"
command: "java -jar main/FitnessEvaluation.jar
   capacity={knapsack_capacity} weights={items_weights}
   profits={items_profits}"
```

Listing 5.3: Complete Agent Configuration

Our implementation of the fitness function.

```
double fitEval(Individual individual) {
   double profits = 0;
   double used_weight = 0;
   double available_weight = 0;

   for (int i=0; i < individual.solution.length(); i++) {
      int bit = Integer.parseInt(individual.solution.substring(i, i+1));
      profits += (bit * this.items_profits.get(i));
      used_weight += (bit * this.items_weights.get(i));
      available_weight += this.items_weights.get(i);
   }

   System.out.println(String.format("Profits: %.2f", profits));
   System.out.println(String.format("Used Weight: %.2f", used_weight));
   System.out.println(String.format("Available Weight: %.2f",
      available_weight));

   double scaled_unused_weight = available_weight
      * Math.abs(capacity - used_weight);
   System.out.println(String.format("Scaled Unused Weight: %.2f",
      scaled_unused_weight));

   double fitness = profits - scaled_unused_weight;

   individual.fitness = fitness;
   return fitness;
}
```

Listing 5.4: Complete Fitness Function

This is the Main method of our *Fitness Evaluation* operator.

```
public static void main(String[] args) {
   // Collect individual from input file.
   List<Map<String, String>> input_data = new ArrayList<>();
   try {
      File inputFile = new File(input_path);
      Scanner inputReader = new Scanner(inputFile);
      while (inputReader.hasNextLine()) {
         String line = inputReader.nextLine();
         line = line.replace("{", "");
         line = line.replace("}", "");
         line = line.replace("\"", "");
         String[] fields = line.split(",");
         Map<String, String> data = new HashMap<>();
         for(String field : fields) {
            String[] key_value = field.split(":");
            String key = key_value[0].strip();
            String value = key_value[1].strip();
            data.put(key, value);
         }
         input_data.add(data);
      }
      inputReader.close();
   } catch (FileNotFoundException e) {
      System.out.println("Provided file was not found.");
      e.printStackTrace();
   }
   Map<String, String> ind_map = input_data.get(0);
   Individual individual = new Individual(ind_map.get("solution"),
      Double.parseDouble(ind_map.get("fitness")));

   System.out.println(String.format(
      "Evaluating individual '%s'", individual.solution)
   );

   // Parse input data and create fitness operator with it.
   // Passed in as arguments like so "capacity=150"
      or "profits=1,5,2,4,3"
   Map<String, String> argVals = new HashMap<>();
   for(String arg : args) {
      String[] keyVal = arg.split("=");
      String key = keyVal[0];
      String val = keyVal[1];
      argVals.put(key, val);
   }

   String profitsStr = argVals.get("profits");
   String[] profitsArr = profitsStr.split(",");
```

```
        ArrayList<Integer> profits = new ArrayList<>();
        for(String profit : profitsArr) {
           profits.add(Integer.parseInt(profit.strip()));
        }

        String[] weightsArr = argVals.get("weights").split(",");
        ArrayList<Integer> weights = new ArrayList<>();
        for(String weight : weightsArr) {
           weights.add(Integer.parseInt(weight.strip()));
        }

        FitnessEvaluation fitEval = new FitnessEvaluation(profits, weights,
           Double.parseDouble(argVals.get("capacity")));

        // Evaluate the fitness of the given individual.
        fitEval.fitEval(individual);
        System.out.println("Evaluated individual with fitness: "
           + individual.fitness);

        // Write individual with computed fitness to output file.
        try {
           List<String> content = Arrays.asList(
                 "{",
                 String.format("\t\"solution\": \"%s\",",
                    individual.solution),
                 String.format("\t\"fitness\": %f", individual.fitness),
                 "}"
           );
           Path file = Paths.get(output_path);
           Files.write(file, content, StandardCharsets.UTF_8);
        } catch (IOException e) {
           System.err.println("Caught IOException: " + e.getMessage());
           e.printStackTrace();
        }
     }
```

Listing 5.5: Complete Fitness Evaluation Operator