1 Requirements Specification

Previous solutions to the problem of automated cloud configuration selection have focused on specific use-cases or application types, and have not provided any associated functional implementation. Our design should be generalisable to any form of application deployed on the cloud, and should be able to recreate previous solutions. This paper should also come with an associated implementation, which can at least perform a Bayesian Optimization-based search for a given docker container containing a batch job or benchmark.

The solution should be capable of:

- Performing optimization algorithms wrapped around some deployment schema
- Specifying cloud configurations based on a collection of input variables
- Provisioning specified cloud configurations from multiple possible providers
- Installing or setting up a container platform such as Docker onto provisioned instances
- Deploying user-specified images onto provisioned instances
- Retrieving logs from remotely deployed containers
- Interpreting logs into single values which are returned to the optimization algorithms

The solution will, preferably, also be capable of running multiple concurrent sample 'jobs' for the optimization algorithms. While many of these requirements are either met, or a means of performing them provided through APIs or other tools, we hope to provide an implementation that combines them into a single generalizable automated cloud selection tool.

We should be capable of effectively replicating previous optimization methods such as CherryPick [1], Ernest [2], or PARIS [3] in our system. We must also ensure that our solution is capable of provisioning instance and deploying applications on machines from multiple providers, and as such cannot rely on any provider-specific API or command-line interface, but will use one of several Infrastructure-as-code tools now available. This is just one of the ways that extending a search space to include multiple providers causes additional problems. To ensure that any optimization algorithm will be effective, we must encode the search space for cloud configurations such that instances are described in terms of a set of descriptive variables. At the time this is written, no universal or cross-provider API exists in the Infrastructure-as-code tool we implemented (Terraform) that allows instance types to be looked up based on their specifications, and so we must develop our own method for describing them for lookup by the optimization algorithms.

2 Design

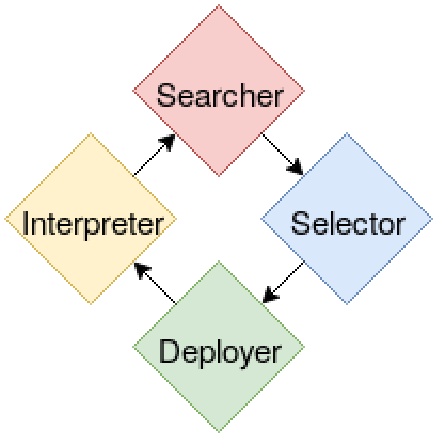

Figure 1: A diagram of the design.

In any optimisation process, the value of some objective function is minimised or maximised by adjusting its input variables or parameters. An optimisation algorithm begin with an initial guess of these variables and iterates through improved estimates until they terminate, hopefully providing an estimated solution. [4] In our generalized case, the optimisation method and objective function are both unknown, but we know that our objective function will always involve selecting some cloud configuration based on the inputs that describe that configuration, deploying some application onto this configuration, and interpreting its performance to give some objective measure.

This process can be broken down into four components: A Searcher, which runs the optimisation algorithm, testing out various inputs in an attempt to maximise or minimise the objective function; a Selector, which interprets the inputs to determine what cloud configuration is being tested; a Deployer, which provisions the machines needed for that cloud configuration, deploys the application, and once it has terminated returns any required logs from it; and an Interpreter, which takes these logs to calculate the objective measure which is returned to the Searcher as the returned value for the objective function. A diagram of this breakdown is shown in figure 1.

It is assumed that in the vast majority of cases, the user would provide their own Interpreter and Selector. Interpreters and Selectors are very dependent on the form these logs will take, and the form the search space will take, and are extremely hard to generalise. For this reason, the modular design of our solution should make it simple for any component to be supplied or replaced by the user. In only some cases should the user need to provide their own Deployer. Applications can often be contained within Docker containers, and aside from occasional setup, for example in multi-node clusters or multi-container applications, a Deployer which provisions a given configuration from a given provider, and then deploys and attaches to a user-provided docker image to collect its logs will be sufficient. Only in rare cases would the user be required to provide their own Searcher. This is because optimization algorithms can be applied to any deployment, with only small modifications necessary in rare cases for specific cases, and an implementation of Bayesian Optimization will be provided.

2.1 Searcher

The Searcher component performs an optimisation algorithm, such as Bayesian optimization, coordinate descent, random search, or exhaustive search, and drives the optimization process by iterating through potential input variables. For each set of these inputs, it take a sample from a single 'job,' where it runs through a single loop of the other three components. The constraints for these input variables must be specified, by describing their type (integer, float, categorical) and limits. A description of how to model cloud configurations into a set of variables is done in the Selector section. Ideally, we want our searcher to be capable of performing multiple jobs concurrently.

Here we describe and compare the design considerations of two important examples of searchers which will be used for the evaluation of our implementation.

2.1.1 Exhaustive search

In an exhaustive search, every possible combination of the inputs is sampled, giving a complete analysis of the entire search space. This obviously takes many samples, $n * \prod_{i=1}^{j} x_i$ where x_i is the number of options for the ith of J variables, and n is the number of samples taken from each configuration. This results in a large or even infinite search cost and time, but is almost certain to return the optimal result, depending on the amount of randomness involved in sampling.

2.1.2 Bayesian Optimization

Bayesian Optimization is an optimization method specifically designed for situations where the objective functions itself is unknown beforehand and expensive to perform, but can have its outputs directly observed through experiments [5]. It models the objective function as a stochastic process, a 'prior function,' and computes its confidence intervals based on samples. Using these confidence intervals and a pre-defined acquisition function it decides future samples to take based on where there it estimates there to be the most potential gain. By this process Bayesian Optimization can find optimal or near-optimal solutions for any non-parametric problem in a relatively small number of samples compared to other optimization methods. In addition, whereas other methods, such as exhaustive search, may handle uncertainty through sampling results from the same inputs multiple times, Bayesian Optimization can incorporate this uncertainty into its estimates, further reducing the number of samples needed.

There are a number of possible prior functions, acquisition functions, and stopping conditions that can be used with BO, and the Cherrypick paper goes into detail on the reasoning behind which options are best for cloud configuration selection specifically [1]. Some notable differences in our case, however, is that CherryPick was specifically focused on batch jobs, where what is measured is simply a function of an instance type's cost and its time taken to perform a given job. Its acquisition function is specified to this purpose, minimizing costs but biased towards configurations which satisfy a soft performance constraint by successfully performing the batch job within a given time. In our case, our acquisition function must be more general, and we will therefore be relying on the user to ensure that whatever objective measure it returned by their Interpreter has already taken into account soft or hard performance constraints such as this.

2.2 Selector

The Selector interprets the variables provided by the Searcher component into the form of an available cloud configuration. Cloud configurations have a number of variables that can describe them, such as vCPU number, memory amount, disk speed, number of instances, instance category, machine type, and cloud provider. The selector must use whatever combination of these is provided and find either the exact or most similar cloud configuration available, passing this information on to the Deployer. In the tools used for our implementation, no cross-provider API was available to directly translate machine specifications into the virtual machine types available from different providers.

For finding an exact match, the Selector can simply lookup the appropriate instance type from a dataset according to the input variables stored as each instance type's attributes. Looking for a closest match rather than an exact one gives more flexibility in how the input variables can be encoded, but means more complicated decisions such as attribute priority must be made, and extra assurances made to not repeat unnecessary samples when multiple sets of inputs describe the same closest input type.

Whether using exact or closest match, it must be decided how to encode cloud configurations into a set of input variables. This problem is further complicated by the fact that constraints for certain inputs may depend on the values of others, as is often the case in cloud computing. Large memory amounts are often only available on machines with more vCPUs, and some providers may offer available configurations others do not. Google Cloud Platform allows users to specify custom machine types, but even these do not allow any possible combination (for example, Memory constraints are tied to vCPU number, and

Instance Type	Provider	Category	vCPU Number	Memory
n1-standard-2	GCE	General	2	7.50
n1-standard-4	GCE	General	4	15.00
n1-standard-8	GCE	General	8	30.00
c5.large	EC2	CPU	2	4.00
c5.xlarge	EC2	CPU	4	8.00
c5.2xlarge	EC2	CPU	8	16.00

Table 1: A possible way of separating instance types into 4 descriptive variables. Providers were either the Google Compute Engine (GCE) or the Amazon Elastic Compute Cloud (EC2)

vCPU number must be divisible by 2).

Despite these problems, clear patterns prevail throughout leading cloud providers, such as separating machine types into categories equivalent to 'Compute-optimised', 'Memory-optimised', and 'Storage-optimized,' each with a set of machines with between 2 and 96 CPUs. In lieu of a cross-provider service to match a given specification to a specific cloud instance type, something which would be outside the scope of this project, these industry-standard categorisations can be used to encode the search space in a reasonable manner. Table 1 shows an example of how we have used these patterns to encode 6 instance types into a set of 3 easily interpreted variables; Provider, vCPU number, memory, and machine category. A searcher tool could easily filter a dataset of this form to find the instance-type for a set of input variables.

In the end, however, the important features and constraints for the search space will differ for each user, and it may be beneficial to run multiple experiments in different search spaces before settling on a final decision for an instance type. While we provide in our associated implementation an example of how to encode and select available cloud configurations for our Bayesian Optimization tool, we think it best to ultimately leave it to the user to design and implement a Selector system that works well for their specific use-case.

2.3 Deployer

The Deployer deploys the user-provided application, batch job, or benchmark onto the selected cloud configuration, and collect any necessary analysis from it. Typically this will involve provisioning the necessary machines from the given provider, followed by deploying the given application onto these machines, and either collecting logs from them or from a networked instance or cluster.

2.3.1 VM Provision

Aside from in serverless computing, the first step of deploying any cloud-based application is likely to be provisioning the virtual machines themselves from a cloud provider. The Deployer should be capable of requesting any virtual machine chosen by the Selector, regardless of provider. This can be simplified using Infrastructure as code (IaC) tools such as Terraform, which offers the ability to codify the APIs from many different providers into declarative configuration files. As long as configuration files and credentials for use by an IaC tool are supplied for each possible provider, then a Deployer can call the corresponding IaC tool to provision the machines. There are several requirements for the IaC tool that is used. As instance type and number of machines will be supplied by the Selector, and therefore cannot be known until the machines are provisioned, either they must be declarable as variables when the tool is run, or the configuration files must be suitable for automated editing just beforehand. Along with this, the tool must support multiple concurrent tasks with their own specifications and outputs, if we hope to allow the Searcher to take multiple samples at once.

2.3.2 Docker Deployment

Once the virtual machines themselves have been provisioned, the application must be deployed onto them. We have assumed that the application being tested is available in the form of a Docker image, and that the machines provisioned operate as either a Kubernetes based cluster or a single instance. While other forms of application or cluster architecture may be used, ready-made Kubernetes clusters are available on several major cloud providers. The modular design allows users to implement their own Deployers to deal with alternative situations.

Both Kubernetes and Docker offer remote APIs. For Kubernetes clusters available on cloud providers, no set up is required, while for single machines the VM provisioner must install Docker and direct it to a public-facing port. The remote APIs can be utilized as long as the VM Provisioner is capable of returning the public facing IP of the provisioned machines, and that security credentials are provided. The Deployer must attach to any deployment, or wait for its completion, and obtain and return any and all logs it produces for use by the Interpreter.

2.3.3 Simulated Deployment

It may be advantageous, especially during debugging or if using a 'closest match' approach to configuration selection, to instead either simulate the response from provisioning and deployment of an application, or to return a previously sampled response. To this end, it may be useful to ensure full logs of any deployment are stored so that, if desired, future calls to 'Deploy' an already sampled configuration for a given application can simply be responded with a randomly distributed value based on previous results from the same configuration.

2.4 Interpreter

The Interpreter must interpret whatever logs and other information is returned by the Deployer, along with the cost of the cloud configuration provided by the Selector, in order to return an objective measure for the sampled cloud configuration. It is this returned value that the Searcher will be attempting to minimise or maximize. We leave it to the user to develop an Interpreter for their use-case. This will likely involve extracting relevant information from the deployed application's logs, and applying constraints based on the time taken or machine pricing.

3 Implementation

Our solution was implemented with a combination of Bash scripts and Python 3. For Selector, Deployer, and Interpreter, a single core Python script was made to call user-specified functions from the three components. This script would be called by a Searcher component for each sampling job, taking the input variables as its parameters, and returning the objective measurement given from that configuration, as defined by the Interpreter. User-provided options, such as what Selector, Deployer, and Interpreter to use, as well as other user supplied variables can be provided in the form of a JSON file that is read and loaded as a Python dictionary by this core script. This dictionary is passed as input to each component, which are able to update it with any information needed by other components in the form of key-value pairs. At the end of any job, this dictionary object is saved in JSON format for later analysis and recovery of previous results.

3.1 Searcher

Exhaustive search was performed using simple shell commands. Bayesian optimization was performed using Spearmint, an implementation of BO written in Python. However, the latest available implementation of Spearmint was found to be outdated and incompatible with the latest versions of various Python modules used in later steps. Because of this, for the sake of generalisability we first updated Spearmint to be compatible with Python 3 and newer versions of its dependencies such as Google Protocol Buffers.

This implementation of spearmint has been made available.[1]

3.2 Selector

A CSV file was created with the following variables for the instance types used in our evaluation:

- API Name - Name of instance type used in that provider's APIs (string)
- Provider - Cloud provider offering the instance type (string)
- CPU - Number of vCPUs (float)
- Memory - Amount of memory in GB (float)
- Category - Category of machine type, such as Compute or Memory optimized, in a consistent form between providers (string)
- Price - Hourly cost of the machine for an On-demand Linux image (float)

The selector loads this dataset as a Pandas dataframe, and filters it according to the input variables provided, and returns the API name and hourly cost for the cheapest option remaining.

3.3 Deployer

3.3.1 VM Provisioner

The first stage of any tested deployment was to first provision the necessary virtual machines from their cloud provider. To this end, a separate folder for each provider was created with a corresponding Terraform configuration file. The configuration files would, when applied using Terraform, provision a number of machines of a given type, and set up a publicly accessible Docker host on each machine. The number of machines, type of machines, and other important variables such as credentials file locations can be specified when the configuration plan is applied, or placed in a single 'tfvars' file in the same folder as our driving Python script. The Terraform plan outputs a timestamp, configuration details, and the public IP, which are used used by later Deployment steps.

It was important that we could run multiple Terraform plans at once, but using a separate configuration setup for every job would lead to extremely large disk usage and overhead from downloading necessary modules into every folder. Because of this, each deployment uses a different 'local back-end,' storing the state files in a separate folder. Output values are only taken from the standard output returned when the

[1] https://github.com/briggsby/spearmint3

configuration plan is first applied, as attempting to get the outputs later can lead to returning outputs from different back-ends when separate deployments attempt to obtain this information at the same time.

We used a Python library python-terraform[2] for our implementation. The public IP address is used with the Python Docker library to deploy docker images onto the provisioned instances.

3.3.2 Ping server

In some cases, a separate server may be used to simulate network requests to a tested application. In these cases, a separate Kubernetes cluster was provisioned beforehand, and its details and credential file locations written into the configuration file. This allows the pinging Docker image to, after the application is deployed successfully on the tested instances, be deployed on the Kubernetes cluster and its logs returned.

3.4 Interpreter

For Interpreter functions used in our evaluation, simple Python scripts were implemented that used regular expressions to extract relevant information from the returned logs. These values, often scores from some benchmark, were then divided by the hourly price of the instance. This is not likely to correspond to any real metric used by a business, as even small advantages over a competitor may lead to a dramatic uptake in users, but was thought to be an effective enough measure to use for evaluating our implementation.

[2]https://github.com/beelit94/python-terraform

References

[1] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, M. Zhang, Y. University, H. Harry Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, pp. 469–482, 2017.

[2] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Nsdi, "Ernest : Efficient Performance Prediction for Large-Scale Advanced Analytics," *NSDI'16 Proceedings of the 13th USENIX conference on Networked Systems Design and Implementation*, pp. 363–378, 2016.

[3] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best VM across multiple public clouds," in *Proceedings of the 2017 Symposium on Cloud Computing - SoCC '17*, (Santa Clara, CA, USA), pp. 452–465, ACM Press, 2017.

[4] J. Nocedal and S. Wright, *Numerical Optimization 2nd Ed.* No. 9781447122234 in Springer Series in Operations Research and Financial Engineering, Springer New York, 2006.

[5] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, pp. 2951–2959, 2012.