ACIT5900
MASTER THESIS

in

Applied Computer and Information Technology (ACIT)
May 2022

Cloud-based Services and Operations

Beyond Bin-Packing

Combining bin-packing algorithms tailored for automated software testing: ROST-algorithm

Martina Rebekka Førre

Department of Computer Science

Faculty of Technology, Art and Design

Acknowledgement

I was hired in Zivid two and a half years ago as a software engineering trainee. Through my work there I have learned a great deal about software testing, continuous integration and the problems that comes with it. I would like to thank my employer, Zivid, for introducing me to the problem area of my assignment. I also want to thank you for the incredible opportunity to learn all I have about CI systems and developer efficiency. Lastly, thank you for being an incredible place to work, and bringing me more motivation to continue working on my master thesis.

I want to thank my supervisor who was always there to guide me in the right direction and put me back on track whenever I discovered new information that made me doubt the path I was on. Thank you for reading through my assignment numerous times to give me feedback along the way. I hope the journey was as interesting to you as it was for me. Finally, thank you for taking the time and making room to be my guidance counselor even though you had filled your quota with students already.

I want to thank my closest friends and family for moral support through the writing process. It was at several occasions very hard to find the motivation to go on, but you were there for me, helping me get back on track. I especially want to thank my mother and my partner who read through big parts of the assignment and gave me feedback on the content.

Oslo, January-May 2022

Martina Rebekka Førre

Abstract

In this project, bin-packing has been analysed as a way of better utilizing resources in a continuous integration system in order to increase its efficiency. The worst-fit, best-fit, next-fit and first-fit algorithms have been investigated in conjunction with an optimal job scheduling approach. The many differences between bin-packing and software testing inspired a solution combining several approaches. The ROST algorithm, short for Resource-Optimized-Software-Testing algorithm was developed to be adapted to software testing. The ROST algorithm is a combination of an optimal job scheduling approach, the worst-fit and the best-fit algorithm where the algorithm changes based on the current bin capacity. The results show that it is just as good or better than the other approaches it has been compared to, and significantly better than the most common solution in continuous integration.

Contents

List of Figures

List of Tables

1 Introduction

The pace of software development has been accelerating since the first digital computers were created, and there is no indication that this will slow down any time soon. New technological improvements lay the foundation for even faster development in the future.

The high pace of software development pressures every technology company to produce high quality products continuously and at a short interval. The competition is fierce and the ability to stay ahead of competitors is crucial to staying afloat in the market. Any lack of quality or slow delivery and the customers will look elsewhere for better products, causing product sales to decrease. This responsibility trickles down within the company and puts pressure on the developers and the pace they can release new features.

The deadline is in an hour. The customers are ready, waiting for the software to become available on the website. Is the product stable? Is it bug -free? Will it be usable in the customers' deployment next week? At the end of the day, if the product isn't good enough, a customer is lost.

Software engineering is a field that focuses on solving real world problems by creating computer systems and applications. A known way of increasing effectiveness in a software team is by increasing the pace of the workflows. Moving a task from the to-do list into the done column is the goal, but there are a lot of steps along the way before a task can be marked as complete. The tasks vary between teams, but very often consist of development and testing.

Testing is an important step for creating high quality products, and while development is mostly manual, many forms of testing can be done almost completely automatically. Continuous integration is a system that automatically builds and tests code changes when they are added to a code repository. This reduces the amount of manual work for

the developers considerably and still ensures that a product is delivered with high quality.

One problem with a continuous integration system is that it requires resources. An application or system should be tested on all combinations of hardware, OS and software that it supports in order to reduce the amount of bugs the customers are exposed to. The continuous integration resources can be hardware or cloud instances and both of these have a cost.

It is a challenge to optimize the resource cost in a system that demands high amounts of resources. One way of reducing costs is to make sure that all the available resources are utilized in the best way. This can remove or reduce the need to acquire more resources while it still ensures that the system is able to run at a high pace.

The term bin-packing comes from packing items optimally into bins. In a software system it is a way of sorting tasks to fit as much as possible into as few hardware resources as possible. Imagine playing Tetris, but not with just two dimension which most are used too, but in several dimensions. Trying to fit the blocks into all the required categories while still filling all the boxes optimally is a challenge. A test in a continuous integration system can have requirements to CPU, GPU, disk usage, memory, operation system or software.

A busy continuous integration system is constantly placing these Tetris blocks of new tests into hosts, creating an interesting resemblance to bin-packing. Nevertheless, the placement isn't optimized to utilize all the available resources, letting resources go to waste and tests take longer than needed. Bin-packing could potentially help balance the placement more optimally, but more research is required to establish the feasibility of this combination.

1.1 Problem statement

The bin-packing concept is an interesting approach when investigating optimal resource utilization in a software testing system. From this, the following problem statement emerges: *Explore the feasibility of applying bin-packing for optimizing resource utilization in software testing.*

The *resources* are the available hardware components in a computer. *Optimizing* in this context means being able to *utilize* as much as possible of the available *resources* by placing tests on the different hardware based on the required resources for each individual test. *Bin-packing* is a way of *optimizing* placement of objects, which in this case is tests. This is done by looking at the test resource usage together with the available hardware to fit tests into smaller portions of the remaining resources. The *feasibility* of using *bin-packing* when placing tests onto hosts in a CI system will be analyzed and discussed to establish if this could improve the hosts' *resource utilization*.

1.2 Document overview

The next chapter is the background chapter which will present the concepts used in this assignment. It will go into details on continuous integration, what it is and how it works. Within this concept, the differences of using cloud instances and on-premise hardware for a continuous integration system will be elaborated on. From there we move over to looking at container orchestration services, what they are, how they work and why they are important for this assignment.

The next concept is bin packing, different algorithms and how it is used in other IT solutions. Finally, the background chapter briefly introduces the optimal job scheduling approach before moving over to the approach chapter.

The approach chapter will explain in more detail how to approach the problem, which resource method is being used and what advantages and disadvantages this approach

brings with it. From there the chapter will explain what the goal of the project is and a little bit about risks and pit falls that might occur along the way.

The results chapter is where we explore the two concepts, bin-packing and automated software testing and how to combine the two. A model is created where several of the concepts described in the background chapter is used and at the end, the solution is tested through a simulation and compared to other approaches.

The discussion chapter goes more into details on what happened during the process of creating the results and what have affected the results in different ways. At the very end, possible future development is discussed as this project should be seen as a part of a bigger project.

1.2.1 Summary of results

In the result chapter, a solution to the resource utilization problem will be presented. Bin-packing was shown not to be a straight forward fit into software testing and the model created therefore exist of a combination of several concepts described in the background chapter. The simulation has shown significant improvement from the solution that exist today, and smaller improvements from pure bin-packing and job scheduling approaches.

2 Background

Before we start looking more closely on the existing research within this topic, let me introduce you to my own motivation behind this assignment and the challenges of testing resource management as seen through my eyes.

In my workplace, the daily meeting for the software team starts at 9.30. Here, all the team members present what they are going to do that day and what they need from the rest of the team to get it done. During my presentation of implementing a new testing pipeline to our Github Action continuous integration system (CI), I said: ”if I need to test run the new pipeline in the CI, I will do it over the weekend”. I didn’t think about it at the time, but there was no reaction from the team regarding me working during the weekend. We all know that the CI resources are limited, and using a lot of the capacity will cause additional and unpredictable waiting time for the rest of the team. When I got to the part of my task where I had to test my code in the pipeline, I was hesitant because I knew the test would run in the CI for at least 2 hours, blocking resources from doing other tasks. This had me wondering what we can do to improve the situation with the limited resources available. The situation isn’t ideal and a change can improve both efficiency and employee satisfaction by being able to run tests immediately and prevent having to do work over the weekend.

2.1 Continuous integration and devOps

Automation is the primary requirement for DevOps, and DevOps’s main concept is ”Automate everything.” (Mohammad, 2020). The concept DevOps is a combination of practises, tools and philosophies to increase an organisation’s ability to deliver high quality products at a high pace. Continuous integration is a way of automating building and testing of code that has been submitted/pushed to a code repository. The goal is to develop reliable and high quality products continuously and quickly. The concept of continuous integration was brought forward by Grady Booch in 1991 (Basu, 2017) to explain how developing using internal releases represent a continuous integration

system, though the term was not used in relation to integrating and testing several times a day. The popularity of continuous integration is growing. A survey based on responses from 5,993 software developers in 2018 shows that 58% of the respondents were using continuous integration, and of those who didn't, 43% planned to use it in the future (Santamaria, 2018). A simple search of the term "continuous integration" in finn.no, a Norwegian application often used for job searches, gave a result of 44 different job ads requesting this competence just in Norway. Some of the ads were also looking to hire multiple people. This shows that the desire to build and improve a continuous integration system is existing.

"Continuous integration is more than a set of practices, it's a mindset that has one thing in mind: increasing customer value"(Meyer, 2014). There are many advantages associated with using a continuous integration system. Bugs can be discovered early due to constant feedback, releases are likely to have less last minute issues, bugs can be tracked down to a particular commit and it enforces practices and discipline for the developers. At this point, it seems that limited time and resources are the only arguments for not implementing a continuous integration system for automatic testing. The setting up of the CI system with triggers, pipelines and tests is considered by many to be one of these time- and resource consuming challenges.

2.1.1 CI in a nutshell: A chain of triggers

Already back in 2016, there were over 40 available continuous integration systems (Hilton, Tunnell, Huang, Marinov, & Dig, 2016). Some examples of available solutions are Jenkins, Harness, CircleCI, Github actions, etc. Messages are sent between different systems or applications to trigger actions, similar to how a human nervous system sends signals between the brain, organs, muscles and skin. One of the main functions for continuous integration systems is receiving signals when changes are made to a common code repository in order to trigger automated builds and tests. The builds and tests are then often run on separate cloud or hardware instances which the main controller communicates with. These instances host one or more applications often

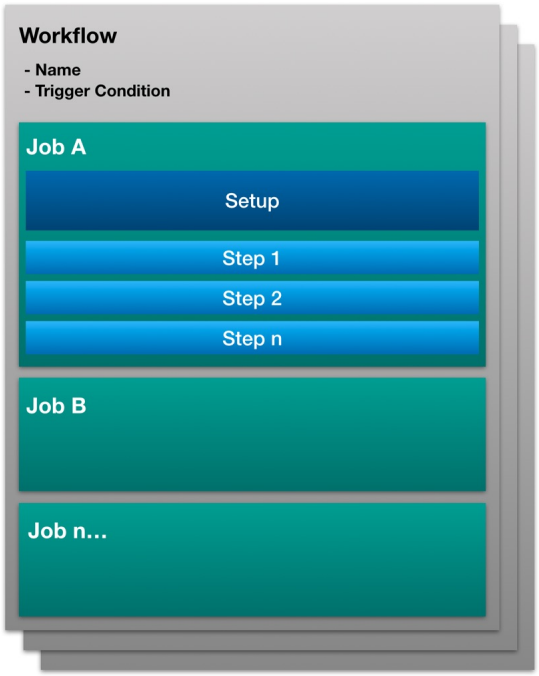

Figure 2.1: Illustration of a workflow and its layers

called runners, executors or agents, though the names vary between vendors. The CI system runs one or more workflows at a time which contains several jobs. A job typically consist of the steps of building and running a single or a set of closely related tests. Figure 2.1 shows how a workflow contains jobs and jobs contains steps. The individual jobs are often spread out to run on their own host/runner/agent.

Github Action, for instance, has named the hosts runners, while Jenkins uses the term agents. One of the difference between these two CI systems, however, is that Github Action can only run one job on one runner, while Jenkins has added an even smaller component within the agent called an executor. This enables the possibility for one host to be one agent, but it can have multiple executors which each can run one job. To have one host run multiple jobs in Github Action, several runner applications have to be installed on the host. And even though it's possible to do, the runners then share environment and disk space. Therefore it is recommended running each runner within it's own virtual machine (VM).

Another functionality such as sending notifications to selected channels on different triggers or events is also a useful feature for the developers to get instant feedback when for instance the workflow fails or a certain action is finished.

2.1.2 Cloud vs hardware resources

A continuous integration pipeline requires resources to be able to run jobs. Some vendors provide the users with runners, but it's also possible to use self hosted runners. Self-hosted runners give the developers full control over operating system, hardware and software, while on the other hand, vendor-hosted runners require less maintenance and setup. Self-hosted runners can also be either on-premise, cloud instances, in containers or virtual machines depending on what the vendor supports. For instance, a company building a hardware product, needs to use self-hosted runners to be able to test software, firmware and hardware using their own hardware.

When it comes to the security aspect of choosing between cloud instances and on-premise resources, there are a few factors to take into account. Typically cloud users don't know the exact physical location of their data and what other data is stored together with their own. If there is a data breach at the cloud vendors, the availability, confidentiality and integrity of many companies' data may be exposed. If, on the other hand, data is stored within the organisation's own infrastructure, the company itself is fully in control of it. Data has six stages within its life cycle: create, store, use, share, archive and destroy and all of these stages need to be secured (Kumar, Raj, & Jelciana, 2018). Resource sharing and multi-tenancy are features of a cloud that makes cloud security more vulnerable. When data is transferred to and from the cloud, encryption is also important to prevent loss of data confidentiality and integrity. These issues present a smaller risk when the entire data life cycle happens within a logical and physical restricted area. However, when a company is responsible for the data of a lot of customers, security will have to be a priority and it is therefore argued that a cloud can be more secure than on premise depending on the security measures within a cloud company.

On-premise hardware is often more expensive for a company as an investment, but over 3 years it will likely be less costly than having a cloud subscription and at year 5, the costs of on-premise is argued to be 50% lower than a cloud subscription (Fisher, 2018). The cloud, however, provides the possibility to scale down services when there is low or no traffic which therefore makes it possible to spend the money only when it is needed.

One of the biggest differences between on-premise resources and cloud instances is how they scale. Both of these can run several jobs within each host, but while it is easy to create, start, stop and delete cloud instances, hardware takes more time to obtain and is harder to get rid of. Using cloud instances therefore creates a more flexible system while on-premise hardware comes with scaling constraints.

In 2020 approximately 81% of businesses had at least one application running in the cloud (Bulao, 2022). While on-premise has less costs long term, cloud solutions are considered to provide more flexibility, scaling opportunities, support services and maintenance. Cloud security, however, can be an argument in both directions depending on the company's needs and the resources they have available for security measures. In the end, both solutions offers advantages and disadvantages and the decision therefore varies based on the individual company's needs.

2.2 Industry

As mentioned, there are a lot of vendors for continuous integration services and they all provide different, yet similar solutions. Some have their own runners at the users disposal, some recommend using cloud instances, some support on-premise hardware through a runner application, but most of them support several alternatives. Whatever resources the company uses, whether it's cloud or on-premise hosts, maximising the utilization of the CI resources is a common desire in order to minimize costs. To be able to maximize resource utilization, auto-scaling runners is required as the system's foundation.

Several CI systems seem to have adapted to the possibility of auto-scaling runners. The most common approaches to auto-scaling in continuous integration systems is:

- Using a container orchestration service
- Using the cloud, often in combination with a container orchestration service

There are several container orchestration services where the most popular today is Kubernetes. Other well known alternatives are Docker Swarm, Apache Mesos and Nomad. There are also several solutions built on top of Kubernetes, for instance ECS (AWS), AKS (Azure), GKE (google), Redhat Openshift, VMware Tanzu, Knative, Istio, Cloudify and Rancher. The main purposes for using container orchestration is to be able to configure and schedule workloads, load-balance containers, allocate resources among the containers and monitor the health of the containers and hosts. Evaluating which host is best suited for a container and evaluating optimal resource usage is the key advantages for better resource utilization.

Kubernetes is used by several continuous integration systems today to manage runners. Github action, Azure pipeline and Jenkins are some examples, while Gitlab uses AWS EC2, built on top of Kubernetes. Table 2.1 shows what auto-scaling methods the different continuous integration providers support. There also exist some non-official approaches, but these are not presented by the vendors themselves.

	Cloud	Container orchestration
Github action	AWS	Kubernetes
Jenkins	AzureVM, EC2	Kubernetes
Azure	AWS, GCP	AKS
Gitlab	EC2	Docker machine

Table 2.1: Some of the CI services that officially support auto-scaling through different methods

What the solutions have in common, however, is that they set hard-coded limits for the number of instances when scaling and base the scaling purely on the number of queued

jobs. The resources each job requires does not affect how many executors fit into one host. The number of runners are therefore scaling freely within the limit when needed. This is a good solution if the company is using cloud instances as the limit can be set based on a maximum cost. When using the cloud, new agents can be created based on the specific resources a job requires. This scaling solution, therefore, works while using cloud resources. When it comes to on-premise hardware on the other hand, the available machines can support the maximum limit of runners which is hard-coded, but it wont support more or less outside these limits if the job requires more or less resources. This, however, may cause the developers to set the maximum limit of runners based on the size of the biggest jobs in order to make sure to not over-allocate resources. Resource over-allocation is a term used when a resource is assigned more work than the capacity of the resource within a given time. The maximum runner limit is set low to prevent problems caused by running too many jobs at once and the result of this is that expensive hardware resources are left unused.

One unofficial approach by Madel, using Jenkins, explores setting resource quotas on CPU and memory for the whole pool of nodes (Madel, 2018). A node can be seen as a host, similar to the runner or agents in the continuous integration system. Figure 2.2 shows a cluster(collection of nodes or a node pool) containing two nodes. It is also possible to set limits on resource usage for individual jobs to prevent them from taking up too much capacity. This solution also uses Kubernetes like most of the other scaling solutions, but has come one step further in the process of optimizing resource utilization, but according to Madel ”Resource Quotas and Auto-scaling Don’t Mix” (Madel, 2018). The reason for this is that when increasing a cluster with one node, the ResourceQuota, which is a variable for the resource limit for a node, would have to be updated manually to make the new resources available to the node pool. So while the node pool can dynamically scale, updating the ResourceQuota still has to be done manually which removes the advantages of automation using auto-scale based on resource usage.

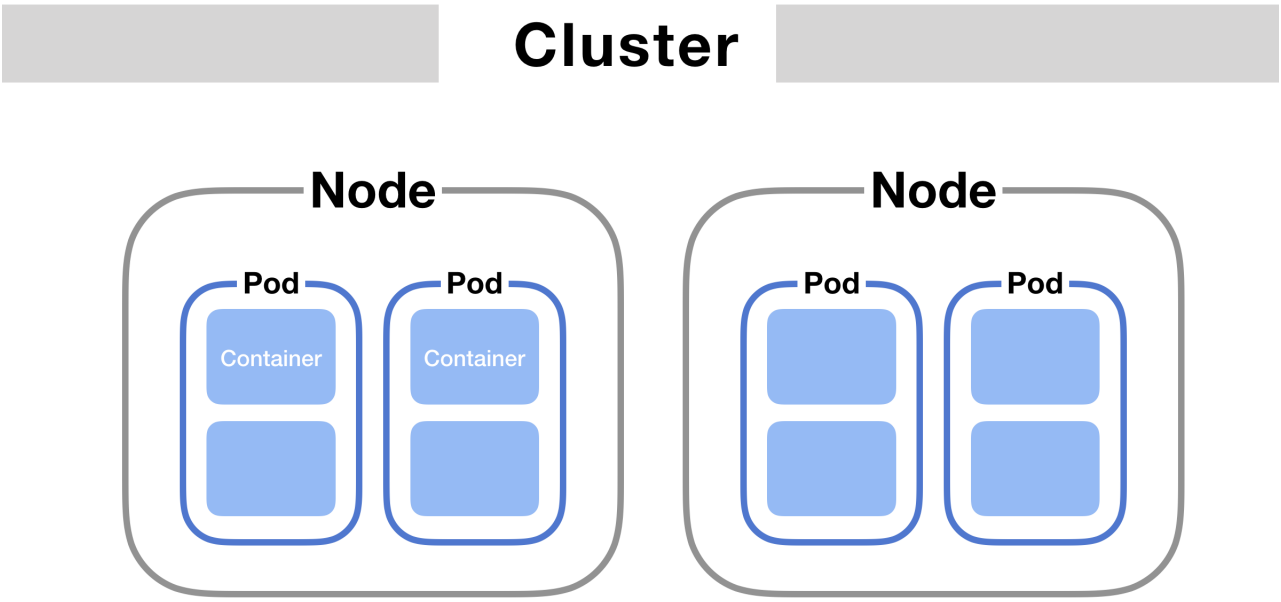

Figure 2.2: Illustration of a Kubernetes cluster and its layers

While cloud scaling-solutions offer more flexibility and therefore utilize the resources well, the on-premise approaches don't consider the underlying available cluster resources. Scaling down while using on-premise resources does not really provide any economic advantages, and the need for scaling then becomes redundant when the limit for the number of runners is hard-coded. However, the need to scale up and down is based more on what jobs needs to be run at any time as they require more or less resources. The solutions for auto-scaling outlined above do not address the problem of utilizing local resources in the best way, and rather scale based on the number of jobs up to a hard coded maximum number of runners. The fact that each job uses different amount of resources is not taken into account.

The cloud is more flexible than on-premise resources when it comes to scaling. It is possible to create new cloud instances to fit the exact purposes of a job and delete it when it's no longer needed. Hardware, on the other hand, requires a company to order or buy new hosts which takes time and manpower to set up once it arrives. It is therefore

more important to utilize all the available resources as optimally as possible.

2.2.1 Container orchestration service and resource management

Kubernetes is, as mentioned, the most popular container orchestration service today. Figure 2.2 illustrates how a typical Kubernetes cluster looks, where nodes represent physical or virtual hosts and pods are the smallest deployable unit, placed into nodes. Pods can contain one or more containers which share storage and network resources (Kubernetes, 2021). In Kubernetes, when a pod is defined, it's possible to specify the resources needed for it, typically CPU and RAM requirements. Kubernetes then reserves the requested resources and also limits the pod by its resource limit. This information is then used to place the pod on a node (Kubernetes, 2022). The scheduler placing the pods ensures that the node capacity isn't surpassed.

When allocating resources to a pod, the resource request variable is set to allocate at least the requested amount. If there are more resources available in the node however, the pod will utilize these resources up to the set limit for the pod. A process that tries to allocate resources over the limit is terminated. CPU and memory are the two main resource types, other than these, non-kubernetes built-in resources are considered as extended resources which can be tied to nodes or clusters. Kubernetes does not need to know what the extended resources are and what they are used for, it only needs to know how much of the resources the nodes have and how much the pods need. This can for instance be used to specify what GPUs a node has and what GPU a pod needs. The resource usage of a pod is reported as part of the pod status, or to other monitoring tools that are available on the cluster (Kubernetes, 2022).

When the current cluster does not have room for the next pod, a new node is required. In a cloud Kubernetes setup, creating and deleting nodes is possible as the cloud is considered to be flexible and scalable, but for an on-premise cluster, this isn't possible. The observation is therefore that Kubernetes does support auto-scaling and it has the

opportunity to place pods optimally onto existing nodes based on available and required resources, but the resource aspect has not yet been adopted by the CI systems.

2.3 The concept of time waste

The goal in a software development project of running automated builds and tests is to free up the developers to be able to do other work and thereby save valuable time. However, running these tests still takes time. After a developer has pushed their new changes to the repository, testing can take a lot of time depending on how many tests and builds are connected to the new version. The developer is, however, mostly aware of how long it will take if the testing starts right away and can therefore plan other work in the meantime. Unpredictable waiting times can be a time-thief as the developer has to check in regularly to see the progress and it might be hard to commit fully to a different task when the time available is unpredictable. When checking the remaining waiting time, it's also easy to lose focus on the current task a developer is working on. If there are no available agents in the CI system and a job is queued, unpredictable waiting time will be the consequence.

The question is therefore: How do we prevent time waste due to unpredictable waiting time? Jenkins writes about solutions to unnecessary queue times in the CI and how to solve it. The first step is checking that all machines are online, the next is to make sure all agents have the correct labels and the third is adding more agents. (Jenkins, 2022). A label is a description of what setup the agent has so that the jobs know if they can use the agent or not. Jenkins can have several executors per agent and CloudBees, a continuous delivery software company, recommends basing the number of executors per agent on the number of CPU cores for the agent. The number of executors should maximum be $nExecutors = nCores - 1$, but the exact number will depend on the use case and job types (CloudBees, 2022). CircleCI has done a cost analysis of machine vs queue cost where they state that allowing queue times of more than 1 minute is like valuing your developers' time at less than a dollar an hour (Bell, 2016). Even for expensive hardware Bell recommend to have queue times of less than 10 seconds.

Bhardwaj writes about addressing the slow performance in Jenkins where memory limitations and CPU bottlenecks are two of the four problems mentioned (Bhardwaj, 2021). Allocating too few resources to a job can therefore also lead to delays for the developers.

In the end, it's easy to plan if we know the number of jobs and the time it takes to finish a pipeline. The number of jobs, however, is unpredictable. How many developers who decides to push their changes at the same time is unpredictable and the number of required runners or executors are therefore also unpredictable. It's the unpredictability that causes the problem. The unpredictable waiting time, the unpredictable job requirements and therefore the unpredictable resource requirements. This is where the automatic scaling comes in. To be able to react and optimize based on what happens at any time.

2.4 The Bin-packing problem

Automatic scaling is the foundation for optimizing resource usage in a continuous integration system. The next step in the process is getting the most out of every resource available. This is where bin-packing comes into the picture. The bin-packing problem is about filling a finite or infinite number of containers, or bins, with different parts, each of different sizes, in order to optimize the amount of content the bins can contain. The bins have a fixed capacity and the goal is to use as few bins as possible. Bin-packing was originally used to fill physical bins as optimally as possible, but has later been adopted to computer systems for instance for backup and placement of virtual machines. The approach has different solutions which focus on different aspects such as weight, cost and speed. The approaches also exist for a single or several dimensions.

Figure 2.3 is an illustration of 5 bins and 9 items where the goal is to use an algorithm to place the items into as few bins as possible. Figure 2.4 shows an example of how the bins could be packed as optimally as possible. The optimal solution would be to add all

items and the result would be a decimal number:
$0.5 + 0.7 + 0.5 + 0.2 + 0.4 + 0.2 + 0.5 + 0.1 + 0.6 = 3.7$. The lowest amount of bins required in this scenario is therefore 4.

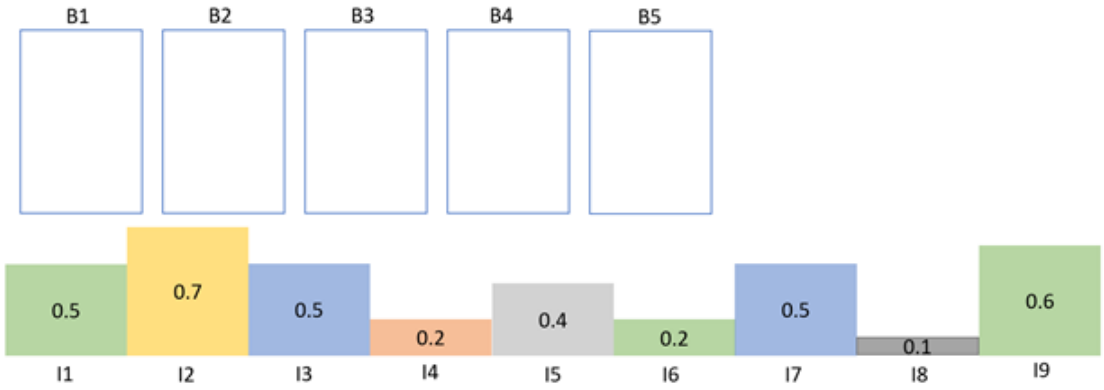

Figure 2.3: Illustration of 5 bins and 9 items. The goal is to place item(I) 1-9 in bins(B) 1-5 as optimally as possible.

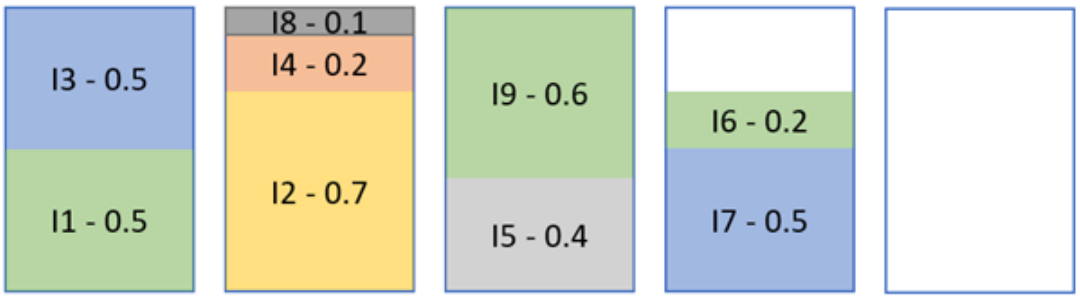

Figure 2.4: Illustration of an example of an optimal packing solution

2.4.1 Bin-packing algorithms

There are a lot of different bin-packing algorithms which vary in both how optimal the results are, and how fast the items are placed overall. The bin-packing algorithms can be divided into online and offline algorithms where the online algorithms aren't aware of the next objects to be placed, and the offline algorithms knows all elements to be placed in advance and can therefore sort them before placing them. The offline algorithms have shown better results in average than the online algorithms, and especially when sorted

from larger to smaller items. This will be discussed more closely later in this chapter in relation to specific algorithms.

The first algorithm presented in this project is the *first-fit (FF)* algorithm. It looks at all bins that already have at least one item in them and puts the next item in the first bin it fits into. If it does not fit in any of them, the item is put in a new bin. The algorithm is fast, but often non-optimal without sorting the items first. Illustration 2.5 shows how the next item should be placed in the first bin it fits into, while figure 2.6 shows the finished placement. The result requires one more bin than the optimal solution in figure 2.4.

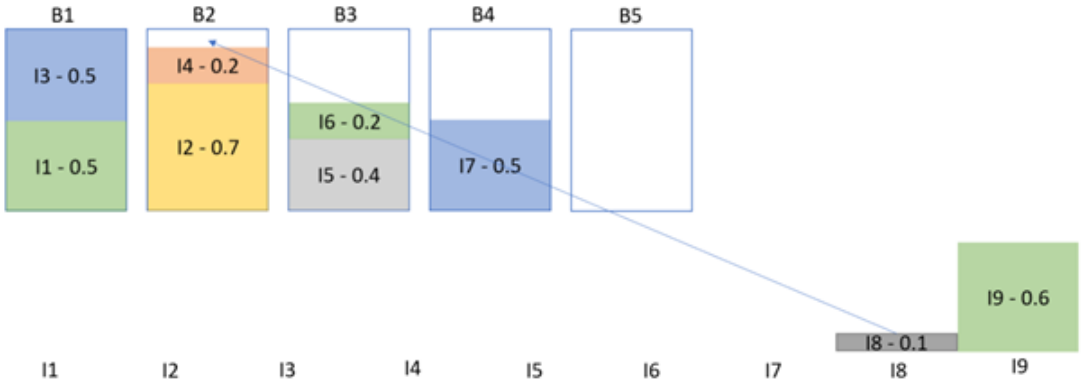

Figure 2.5: Illustration of the first-fit-algorithm after placing the first 7 items

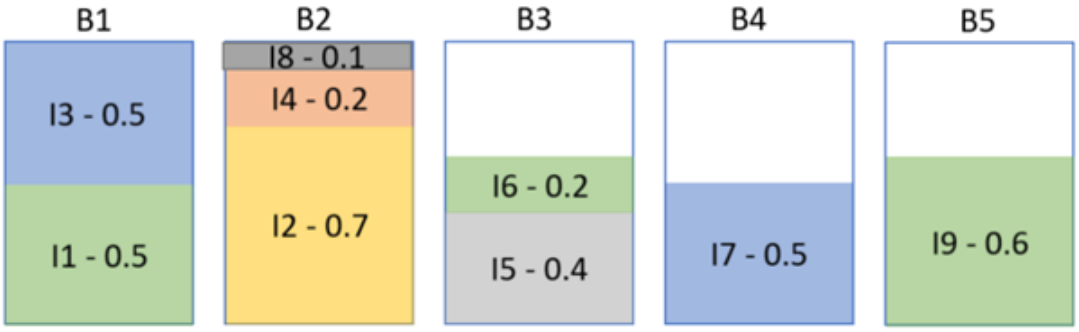

Figure 2.6: Illustration of the first-fit-algorithm after placing all items

This placement can also be demonstrated using a script. The following is pseudo code for the FF-algorithm:

```
def first_fit(all_items: list, all_bins: list) -> int:
    for item in all_items:
        for bin in all_bins:
            if item fits in bin:
                place item
                continue
        all_bins.append(new_bin)
        place item in new_bin
    return len(all_bins)
```

The next algorithm, the *best-fit (BF)* algorithm tries to fit the next item in the bin which leaves the *minimum* amount of space after placement. This often gives better results than the first-fit algorithm as the goal is to fill the bin all the way to the top using the next item. The order of the items do however affect the results. The given order of the items in the case described in figure 2.3 gives the same results when using FF and BF while an other ordering could give different results. The algorithm in this case therefore also results in the usage of 5 bins.

This algorithm can also be implemented and tested using a script or program. The following is the pseudo code for BF algorithm:

```
def best_fit(all_items: list, all_bins: list) -> int:
    for item in all_items:
        minimum_remaining_space = Inf
        fitting_bin = None
        for bin in all_bins:
            if bin.space - item.space. < minimum_remaining_space:
                minimum_remaining_space = bin.space - item.space
                fitting_bin = bin
        if fitting_bin != None:
            place item in fitting_bin
        else:
            all_bins.append(new_bin)
            place item in new_bin
    return len(all_bins)
```

The *worst-fit (WF)* algorithm is almost the opposite of the best-fit algorithm. Instead of placing the item in the bin which leaves the least space, it is placed in the bin leaving the most remaining space after placement. There is also a version of it called the almost worst fit which puts the item in the second worst bin. This algorithm can be slow as it has to evaluate every bin each time a new placement is done. This algorithm is often used for memory allocation in computers to evenly place items out and therefore give more resources (in this case memory) to each task.

All of the above algorithms can also go into the category of being an *AnyFit* algorithm. This means that while the next item can fit in one of the already open bins, a new bin cannot be opened. Assuming that the worst-fit algorithm isn't an AnyFit algorithm, the result would be starting by placing one item in each bin as shown in figure 2.7. Figure 2.8 however illustrates the results of the worst-fit algorithm assuming it is an AnyFit algorithm. The WF-not-AnyFit algorithm will require as many bins as there are items, if the number of bins are infinite, while adding the AnyFit constraint, the algorithm require 5

bins, in this case, the same as FF and BF, though with more space left in each bin.

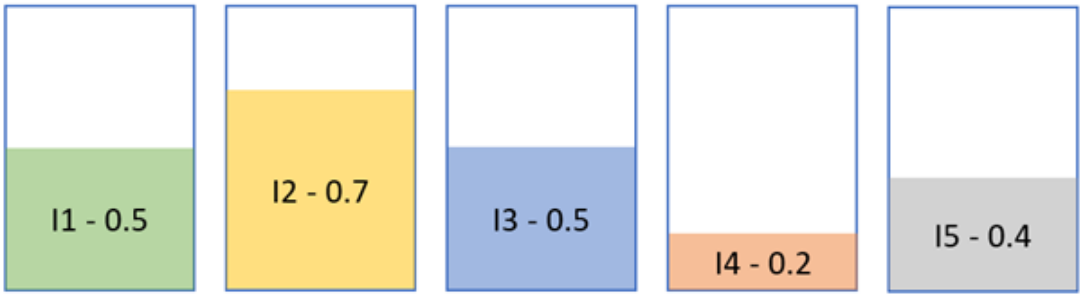

Figure 2.7: Illustration of the worst-fit-algorithm results assuming that it's not an AnyFit algorithm and placing the first 5 items.

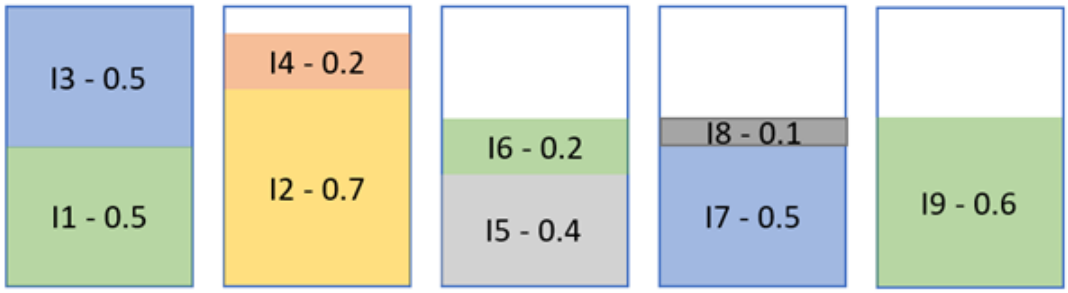

Figure 2.8: Illustration of the worst-fit-algorithm results assuming that it is an AnyFit algorithm.

The *next-fit (NF)* algorithm checks if the next item fits in the same bin as the previous item and if not, puts the item in a new bin. Once a new bin is opened, the previous ones are not accessed again. This is a fast algorithm as it only has to loop over the bins once, but the fact that it can't go back to previous bins will cause empty space where smaller items could fit. This approach can be compared to how some people may pack their grocery bags, when the next item does not fit in the bag, put it aside and start a new bag. This however shows poor results. Figure 2.9 shows the result of placing the items using the NF-algorithm resulting in a total of 6 bins. This is worse than all the previous results except for the worst-fit when not adding the AnyFit constraints.

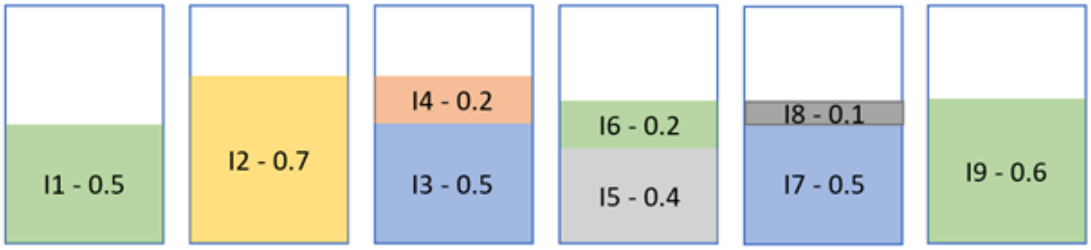

Figure 2.9: Illustration of the next-fit algorithm results

First-fit decreasing and *best-fit decreasing* are offline algorithms which start by sorting the input sequence by size before placing the items in order to be able to place the biggest items first. It then follows the BF or FF approach. This is considered a more optimal solution than a simple FF and BF algorithm, these, however, require that all items are known before placing them. Figure 2.10 demonstrates the advantages of beginning with bigger items before the smaller while figure 2.11 shows the opposite sorting.

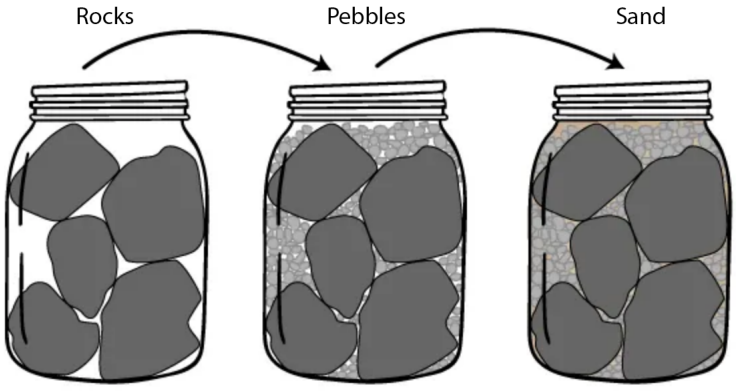

Figure 2.10: Illustration of jars used to demonstrate that placing bigger items first, leaves room to fit smaller items in between (Buggy, 2020)

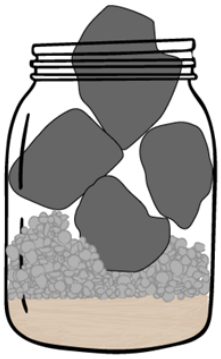

Figure 2.11: Illustration of a jar demonstration that when the smaller items are placed first, the bigger items wont fit (Buggy, 2020)

2.4.2 Bin-packing in IT

Most bin-packing algorithms are developed for placing physical items into physical bins and can therefore leave spaces between the objects that are placed. This isn't always the case in computing, where the resources not used are available no matter the location. Another difference is the way the axes are defined. Figure 2.12 and 2.13 show a typical visual representation of bin-packing in 2 and 3 dimensions. In computing, on the other hand, boxes can't be stacked, as this won't clearly represent the resource usage if we define resources along the axes. Therefore, a more accurate representation will look like figure 2.14 and 2.15. This is called vector bin-packing or multi-capacity bin packing. Regular bin-packing can typically be compared to filling up a truck with boxes, while vector bin-packing can be used for instance in VM placement in the cloud, in the case where resources cannot be over-allocated. Over-allocation can be useful in situations where the resource requirements vary a lot or only require the resources in shorter periods of time. The resources can then be used by different processes at different times and therefore utilize resources better, like for instance operating systems are doing. Nevertheless, it can slow down processes if they have to wait for the resources to be available. An optimal solution in vector bin packing for one bin represented by a two- or three-dimensional vector $\vec{B}$ would be fitting n items represented by vectors $\vec{v}$ so that

$$\vec{v_1} + \vec{v_2} + ... + \vec{v_n} = \vec{B}$$

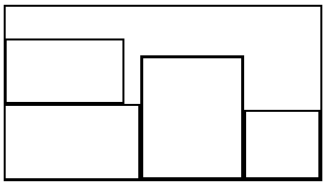

Figure 2.12: Bin-packing in 2 dimension

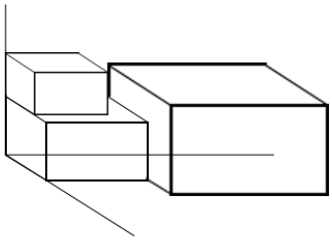

Figure 2.13: Bin-packing in 3 dimensions

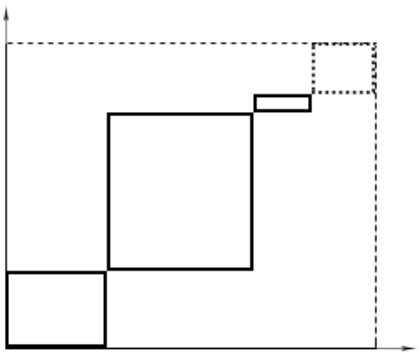

Figure 2.14: Vector bin-packing in 2 dimensions

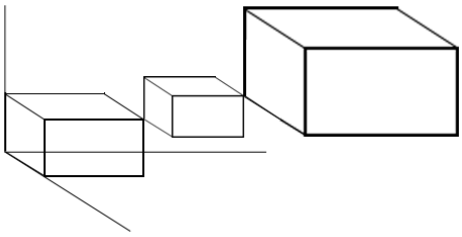

Figure 2.15: Vector bin-packing in 3 dimensions

To establish the dimensions of a bin in bin-packing it's important to consider the bottlenecks as they are different from ordinary boxes. The 5 most common bottlenecks of computing, according to Schultz (Schultz, 2017) are:

- Network speed
- CPU
- Memory
- Disk
- Installed software

Bin-packing has been used for a few years in virtual machine placement onto hosts in the cloud to optimize resource usage and lower power consumption in, for instance, data centers (Karthik, Sharma, Maurya, & Chandrasekaran, 2016). Kumaraswamy and Nair found that the FF, FFD and the Max-Min algorithms achieve maximum performance while the NF performed badly for virtual machine placement (Kumaraswamy & Nair, 2019). The Max-Min algorithm was created because it was observed that the existing algorithms were either fast or achieved optimal placement, and the goal was to have one algorithm that had both. This algorithm is a good example that emphasizes algorithm efficiency, however, the speed of the algorithm itself is not seen as a critical factor at this point.

There is a difference between regular bin-packing and VM packing called the VM packing problem which occurs when two items together require less space than the two VMs individually. This is typical when it comes to memory allocation as some items can share parts of the memory. Jin, Pan, Xu, and Pissinou consider CPU, memory and power consumption as the three deterministic resources and bandwidth as a stochastic resource (Jin et al., 2012). Bin-packing has also been used by Rawitz and Patt-Shamir motivated by network scheduling to ensure quality of service, however they mention that their approach is applicable to other use cases as it has a general formulation (Rawitz & Patt-Shamir, 2012). Szoke has also used bin-packing as a way of optimizing planning of agile releases (Szoke, 2010).

Bin-packing is usually about placing all items in the least possible number of bins. The opposite approach is called maximum resource bin-packing where the number of bins used is maximized to minimize the number or size of the total items. In IT this can be a good approach for distributing items across available hosts to utilize all the available resources. Boyar et al. however sets one of the constraints for this approach to not opening a new bin if the current item fits in one of the opened bins (Boyar et al., 2006). Boyar et al.'s approach can be achieved using the AnyFit version of the WF algorithm. Nevertheless, the approach with this constraint does not accurately solve the problem of maximum resource utilization by spreading the items across the available bins and the not AnyFit WF-algorithm can be a better approach to utilize all resources or bins when they are limited.

2.4.3 Measuring the results

The goal of this assignment is to explore how the resources available can be utilized better which should lead to a more efficient CI system. When focusing on several dimensions or variables during bin-packing, finding the best way of measuring the results can be a challenge. Kumaraswamy and Nair suggest measuring the bin-packing algorithms based on the number of servers/hosts required and the CPU utilization of each server (Kumaraswamy & Nair, 2019). All the virtual machine placement techniques

analyzed in their paper are designed to achieve efficient CPU utilization. de Niz and RajKumar looks into bin-packing software components onto hardware to ensure quality of service while reducing the amount of resources used. This approach partitions software into smaller pieces and then considers the processor as the bin and shows that a significant reduction of bins is possible (de Niz & RajKumar, 2006).

2.4.4 A typical hardware setup

To be able to optimize the resource utilization, it is important to start by observing the resources that are available. The job completion speed is an important focus in a CI system and there are two main hardware components collaborating to increase the speed of a computer, the CPU and memory. According to ArsTech the most sold CPUs on amazon in the first quarter of 2020 has 6 cores, but the top 10 list varies from 4 to 12 cores. The median of the processor cores on the list is 7, as both 6 and 8 cores are highly represented in the list by 4 instances each. The prices listed on amazon at that time was for the 4 core CPU slightly below 100$, the 6 core CPUs had a range between 120-211$, 8 cores between 170-400$ and the 12 core CPU was listed to 470$ on amazon in early 2020 (ArsTech, 2020). Horowitz from HP recommends at least 6 cores for engineers, data analysts and video editors (Horowitz, 2020). Other than cores, clock speed is also important to be able to finish each task more quickly. A different employee of HP, Sirois, recommend a clock speed close to 4GHz for intensive computing (Sirois, 2018).

Besides CPU, the memory, is as mentioned, also an important hardware component to increase efficiency of task completion. Kingston, a hardware component producer, recommends at least 4GB RAM for a computer, minimum 8GB for gaming or professional usage, while a high-end gaming system and workstation is recommended to have 16 or 32GB memory (Kingston, 2019). The RAM speed of different best-selling memory cards mostly vary between 3200Mhz and 3600Mhz, but a few listings is down to 1600Mhz (Amazon, 2022).

As mentioned in the continuous integration chapter 2.2.1, the container orchestration service Kubernetes has the possibility to scale based on resource limitation and requests. The scheduler is able to use bin-packing behaviour by looking at the requested resources and capacity, but which bin-packing algorithm the default Kube-scheduler uses is not specified. There is, however, a Github repository which provides a Kubernetes scheduler called best-fit-scheduler, which uses the BF algorithm.

In IT, CPU, memory, power consumption and bandwidth are considered bottlenecks and can be evaluated as dimensions in bin-packing. A problem with this is that the size of items may vary *after* their placement. This is something that isn't typically taken into account in traditional bin-packing approaches. Another difference is that in software testing, items are also being removed after a test is finished, opening room to place new items.

2.5 Optimal job scheduling

A none-bin-packing approach to the problem of better placement of test jobs, is called optimal job scheduling. This approach takes job removal into account. Speed is introduced as a variable based on processing power where the goal is to finish a collection of tasks/ jobs as fast as possible. This can be approached as a single-stage jobs problem or a multi-stage jobs problem, where a single-stage jobs only has independent tasks and a multi-stage job has several tasks that has to be performed in a specific order. As the CI itself takes care of multi-stage scheduling by not queuing jobs to agents before all required jobs are finished, the scheduling problem comes down to the single-stage job approach. This can again be divided into four different problems:

- Single machine scheduling
- Identical machine scheduling
- Uniform machine scheduling
- Unrelated machine scheduling

The first one, single machine scheduling, uses only one machine to finish the tasks. The second one, identical machine scheduling is relevant if all available hosts are identical, where there's no advantage to choosing one machine over another when there is no tasks already running on any of the machines. The uniform machine scheduling has several different machines, where the completion time for a job J on machine I is based of the speed of machine I = S_I and the run time P. The job J run time on machine I is therefore given by $P_{I,J} = P_J/S_I$. The last, unrelated machine scheduling, on the other hand, has no relation between values of $P_{I,J}$ for different I and J. In a CI context, the job completion time is based on both the length of the job and the speed of the agent, leaving us with the uniform machine scheduling problem.

The job scheduling approach only runs one job on one machine or processor at a time, but opens up space for new jobs once the previous job has finished. This can fill in the missing piece of bin-packing where items aren't removed along the way to leave room for new items. The restrictions to run one item on one host at a time, however, does not fulfil the requirements of optimizing the resource utilization as tasks often require less than a whole machine, but can require more than one processor. The timing and item removal still makes job scheduling partly relevant.

There are different algorithms within machine job scheduling and the goal of the algorithms can either be to minimize the average completion time, or minimizing the maximum completion time. The SPT(shortest processing time first) first sorts the jobs based on length and then assign the job to the machine where it can finish as early as possible. The SPT algorithm minimizes the average completion time, but is created for the identical machine problem. Other approaches have the same goal and usually focuses on placing the next job on the fastest machine or places the longer jobs on the faster machines. Nevertheless, the approaches are different because there are more factors to take into account when the machines aren't identical. Especially in an unrelated machine scheduling problem, each job has to evaluate how fast they will finish on each machine before placement.

In job scheduling, finishing all jobs as fast as possible is the goal. This differs from bin-packing where packing optimally is the goal. Both of these approaches have been explored in the IT world, but there has been few to no documented uses of bin-packing within a software testing perspective. Optimal scheduling, on the other hand, is an obvious focus within the CI world as fast task completion is a strong pulling force. Even though Kubernetes supports a form of bin-packing and most CI auto-scaling systems use Kubernetes or a solution that is based on or similar to Kubernetes, the bin-packing and resource management solution hasn't been integrated into the CI systems yet. The advantages of using bin-packing should be explored further to evaluate if the approach could grant advantages to the existing CI systems.

3 Approach / methodology

The literature review has given an insight to existing solutions, how they work and a brief understanding of the differences between the concepts. Furthermore, to decide on the correct path to take, it's important to go back to the problem statement: *Explore the feasibility of applying bin-packing for optimizing resource utilization in software testing.*

The goal is to explore the feasibility of using bin-packing in system testing and explore if bin-packing will impact resource utilization and whether the system will be able to subsequently run more tests simultaneously. If the bin-packing approach seems to improve the resource utilization, a further goal is to explore if it is possible to implement it.

To explore, in this context, is to analyse how the concepts bin-packing and software testing fit or doesn't fit together and then design a model from the discoveries. The model can be tested through a simulation, where the results can be compared to other existing solutions. This is often an iterative process of designing the model, creating the prototype and discovering new paths which bring the process back to the literature. Decision that has to be made can be discovered during the prototyping phase which leads back to the model or new problems and ideas can lead all the way back to the drawing board. The goal isn't final when heading out on this journey and this has to be kept in mind while going further in the process.

3.1 Research methods

There are several research methods that exist where the exploratory and comparative resource methods are among them. The exploratory research method is based on exploring a problem that isn't yet clearly defined. The approach is open to explore ideas and paths that appear along the way. The research doesn't necessarily reach a conclusion as the goal can keep changing based on what is discovered along the way. The data that is produced will also vary and can be hard to compare and analyze.

Another disadvantage of this approach is that it can be hard to determine how to handle new information that appears during the research period. New information can trigger a desire to change direction completely, or it can be tempting to not mention the find at all. However, this is part of the process, and the conclusion doesn't have to go in favor of the original statement or idea that the assignment originally started with.

A comparative research method has the goal of comparing the research object to others to discover similarities, differences, advantages and disadvantages with an object. This approach seeks to reach a conclusion to the specific problem or question that was asked. This could for instance be a project about comparing existing research on a topic to find the best approach among the possibilities. Comparing different continuous integration systems is an example of a comparative research. This can be a time consuming research method as many advantages or disadvantages doesn't appear until the system is pushed to the limits and implementing several systems into a realistic scenario is time consuming.

This project started with an assumption of resource shortage in a continuous integration system which from there was developed into an idea that the resources available could be utilized more optimally. It was important to go in a concrete direction when searching for answers and bin-packing was an area that seemed promising. The advantages, or potential, for optimizing resource utilization in software testing systems using bin-packing had to be researched before one could begin exploring if using it with continuous integration was possible to do. This idea doesn't fit into a comparative research method as the approach to compare to doesn't yet exist. This is an area that needs to be researched by exploration, therefore putting this project in the category of exploratory research.

Celiku has written a paper about automated canary deployments, which has chosen an exploratory approach. The reason is that there isn't a lot of research conducted on the topic (Celiku, 2021). Both software testing using continuous integration and

bin-packing are topics that are well known and researched, but the combination of the two hasn't been explored, making this a new field to investigate, arguing towards an exploratory approach.

An important principle in the project is to stay close to reality and not force concepts to work with each other just to stay true to the concepts themselves. In this case, the project has to follow the principles of software testing even if the bin-packing concept doesn't completely fit. The overall goal is to make CI systems more efficient, not forcing it into a bin.

When playing video games it is often tempting for many, especially the completionists, to start by taking the wrong path to find all the loot and know that every corner has been explored. The point is to make sure to not miss anything. Unfortunately, writing a short assignment closes many of these doors as the time is very limited and to even get close to the target, the goal has to be selecting the right path as often as possible to avoid dead ends and lost time.

3.2 Project outlines and delivery

In this approach I have to make an assumption that lack of resources is an existing problem, secondly I have to assume that bin-packing for software testing will improve the situation. It has already been shown that it can optimize VM placement (Jin et al., 2012) and thereby reduce resource usage. Perhaps it can optimize software test placement as well.

The research process will start by analysing the two concepts, bin-packing and software testing using continuous integration, and find the similarities and the differences. The goal is to create a model which combine the two concepts in a way that still stays close to reality from the software testing perspective, but how, is currently undefined. The model will be a description of a possible solution to the problem of combining the two concepts: bin-packing and software testing. This will show if the

concepts do or do not fit together. Several bin-packing algorithms will be evaluated: the next-fit, best-fit, first-fit and worst-fit algorithms as well as an optimal job scheduling approach.

The next step is to make a simulation to test the model and use this to determine if bin-packing does what is expected, help utilize the resources better, making a continuous integration system more efficient without adding more hosts. These things will be done in the next chapter of this project. The results of the model will be compared to other approaches and evaluated based on evaluation criteria that will be set in the result chapter. The simulation will be a python script, creating bins and items, then placing the items in the bins using different approaches: the model, different bin-packing algorithms, an optimal job scheduling approach and an approach similar to how CI-systems do it today. The results from the different approaches will be compared and used to decide if bin-packing improves the resource utilization of a continuous integration system. This project will not implement a final solution in an existing CI system, but explore the potential of combining the concepts.

This project should be seen as a part of a larger project. This is just the first pieces in a bigger process and the goal for this project isn't to solve the problem, but to push it in a direction and create a foundation to be able to implement a prototype later. All questions might not be answered after finishing this project, but a direction to further explore will be set.

3.3 High risk, high reward

Even though an exploratory assignment can lead to interesting finds, this approach also has risks. Is there enough time, will the process reach a currently unset goal, and will the project go in the direction intended from the beginning? These are all questions that cannot be answered before the journey is on it's way or close to the end. Nevertheless, there are some measures that can be done to reduce the risk. Weekly guidance meetings to discuss progress, the direction, the model and brainstorming can help keep

the project on track, ensure timely deliveries and help avoid wrong turns. An active writing process from the beginning is also a measure taken to ensure that content is created continuously while it also ensures focus on the assignment which keeps the brainstorming going. Even though there are a lot of uncertainty to the project, these measures will ensure that the risks can only cause limited damage. There are a lot of unanswered questions, yet, the direction is set and the process is ready to begin.

4 Result

Now that the direction has been set, the research process can being. This chapter will evaluate bin-packing as an approach for optimizing the resource usage in a CI system. Different algorithms will be discussed and the goal is to find an approach that will fit well within existing CI system solutions and the restrictions that comes with it. The goal is to create a model where the concepts fit with each other and a simulation to test the model and evaluate it. If this model is implementable will also be discussed.

4.1 What is a bin and what is an item?

The first step towards establishing if bin-packing could be feasible for software testing, is to explore how different bin-packing algorithms could affect item placement into bins. First we have to determine what a bin is to be able to describe the characteristics more precisely. A bin in our scenario has dimensions based on hardware resources that can be seen as potential bottlenecks in the system. To achieve the goal of optimizing software testing resource usage, these dimensions have to be the most critical resources in a hardware setup. If a critical bottleneck isn't defined as a dimension in the bin, the resource can be over-allocated which again can cause errors unrelated to the testing itself. Each hardware resource represents its own dimension in the bin and each dimension must be taken into account when placing items. The bin dimensions will be discussed more closely in subsection 4.3.

What an item is should also be decided as this will affect the size of the building blocks that are placed inside the bins. Figure 2.1 in the background chapter, which illustrates the content of a workflow, has several possibilities for what can be selected as an item. The most optimal choice is to use the smallest possible component as an item. It's much easier to fit a larger volume of sand into a box, than larger rocks. A step is often considered the smallest piece of a workflow. It can, however, contain a script running multiple actions, but the smallest component recognized by the CI are steps. If a step is selected as an item, every step has to be independent of each other, this can be logical

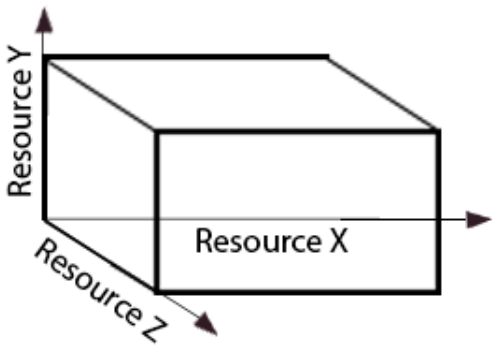

Figure 4.1: A visual representation of a three-dimensional bin where the arrows represent one dimension or resource each. The max capacity of one resource is in this illustration the corner of the bin on the resource axis.

if no initial setup is required. This, however, is rarely the case, which brings us to the next, and more abstract CI component, a job. A job contains several steps which often starts with an initial setup of installing or building the resources required to execute the steps. Splitting steps with the same requirements into individual ”items” will require installing and building the same setup several times which can be more time consuming and resource-heavy than running the steps themselves. It is therefore more natural to run the steps together with the same setup compared to running the individual steps on several hosts. A job is therefore considered as the most natural way to represent an item for bin-packing.

4.2 The ”best-fit” algorithm for CI testing

There are a few other decisions to be made in the process of building a model, where one of the most important ones is which algorithms to use. The two most natural algorithms to start with is the first-fit and the best-fit algorithms which are both relatively straight forward to implement and can be adjusted to the number of dimension and also to a vector bin-packing approach. The next-fit algorithm, however, seems less relevant

as it closes a bin when the next item doesn't fit and the algorithm prioritizes fast placement over optimal placement. In software testing, jobs are removed after they are finished and more items can be placed. The number of bins is also finite, and when the next item can't be placed in the last bin, the algorithm stops. The algorithm can be tweaked to start over at the first bin after finishing the last, but when all the bins are full, the algorithm will continue to iterate over the bins creating an infinite loop until the next items fit in any bin which might require a lot of processing power. The focus on optimizing the algorithm speed over optimal placement leads to the assumption that the NF-algorithm isn't optimal for software test bin-packing. The worst-fit algorithm, which is rarely considered in bin-packing due to it often leaving a lot of empty space in each bin, has interestingly an advantage in IT. The items can use more resources while other items don't need it and spreading items between the available bins can reduce the required run time. This algorithm should therefore be considered for our project.

An online algorithm never knows the next item in the queue, while the offline algorithms know the queue in advance and sorting the queue in advance has given better results. An offline algorithm is therefore preferred in regular bin-packing, but the decision between online and offline algorithms firstly depends on if it is even possible to use an offline algorithm within IT. Our queue is based on the input flow of jobs from the CI system. How the test flow looks is based on how often the developers trigger the pipeline, how many pipelines there are and how many resources are available to run jobs. The jobs in a single pipeline are known in advance which makes the queue predictable. The pipeline can therefore be seen as a puzzle where all the pieces are known in advance, and the job is to put them in the correct location. The problem occurs when there are multiple pipelines or multiple developers that trigger pipelines in the CI at the same time. When several pipelines use the same resources, the input flow will be unpredictable and jobs from different pipelines will, instead of puzzle pieces, be more similar to Tetris blocks. There is some predictability on what is next, but if a new pipeline is triggered, unexpected jobs start to appear, and if a pipeline fails in the middle, jobs that were expected, do not appear at all. As offline bin-packing algorithms consist of

sorting items before they are placed, it is still possible to use this approach. If we continuously sort the tests whenever new items enter the queue there is some resemblance to offline bin-packing.

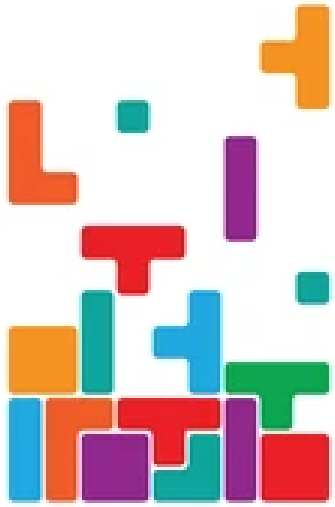

Figure 4.2: Tetris pieces illustrates jobs, but they may come from different workflows, therefore being unpredictable. (Yalcin, n.d.).

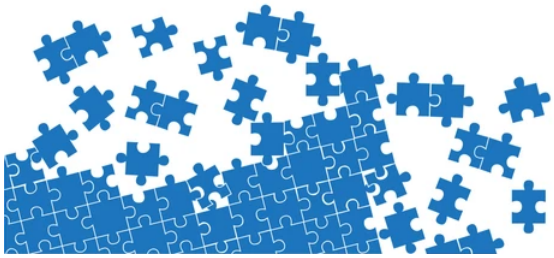

Figure 4.3: A jigsaw puzzle where each piece represent a job in the complete puzzle which represent a workflow. The puzzle pieces are predictable. (srisuk, n.d.).

4.2.1 Choosing the algorithm

When evaluating if it is possible to use offline bin-packing algorithms, it's also important to ask if it should be used in this context. As mentioned, offline algorithms are preferred in regular bin-packing, but what about in a CI system? If there always are resources available so that tests can be placed almost instantly after entering the queue, then there wont be many, if any, queued items to sort. If there is waiting time however, it might be more relevant to sort the items before placement. One problem with the decreasing offline algorithms, however, where items are sorted from bigger to smaller, is that smaller jobs may be continuously put at the end of the queue and therefore never be able to run.

In a continuous integration system, the order is important based on other factors than size. If jobs have different time-use and importance, they may already be sorted in the pipeline and moving them around can cause reduced efficiency. Placing a test that often fails early can for instance be helpful as the remaining pipeline will stop and thus reduce the amount of jobs that have to run. Jobs can also depend on each other, forcing one job to run after another. Offline algorithms and sorting items before placing them has shown to improve approximation guarantees (Wikipedia, 2022), but in the end, sorting CI jobs based on size before placing them onto instances doesn't seem to be the optimal solution in real life, but this doesn't exclude other sorting methods.

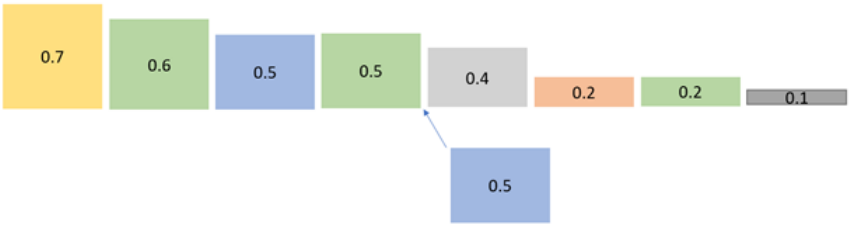

Figure 4.4: Illustration of a sorted item queue, sorted by size. When a new item enter the queue, the items which are smaller than the new item are moved further back in line

The amount of time a bin-packing algorithm uses to decide item placement is often an aspect discussed when it comes to deciding which algorithm is the best. The time each algorithm requires is usually just based on how many items are to be placed, but in a CI system, where the box sizes vary, this has to be taken into account as well. The calculation is, however, run on the controller machine in a CI system, which most commonly does not run jobs and will therefore not use resources from the resource pool. The algorithm processing time is also predictable based on the algorithm selected or is at least deterministic over time. The calculations are also assumed to be fast when using a decent machine today. This leads to the assumption that the algorithm time is negligible in a CI system when selecting the best algorithm.

Even though the time an algorithm uses is assumed negligible, time to completion for each job is an important factor within CI-testing. One thing that is very different between

bin-packing of CI-jobs and regular bin-packing is that in CI systems items are being removed after a job is done, making software testing more dynamic than regular bin-packing. Therefore, how quickly a task can finish on a machine will affect how efficient the CI is overall. This brings timing in as a concept and placing tasks on specific hosts can either slow down or accelerate the overall process. When a task is done, it leaves more room for new jobs to be placed on the hosts. Over time the individual job time can be observed and used to make projections on how long it will take before tasks finish and therefore do an even better evaluation of where it is best to place a new job. This, however, requires a very complex algorithm beyond pure bin-packing if more than just the time should be accounted for. The variables used in the algorithm has to be adjusted over time based on the observation of how long tests run for and how this affect other jobs.

4.3 Selecting bin dimensions

After deciding on which algorithms to use, the dimensions of the bins have to be set. How many dimensions should be evaluated depends on what resources are most important in a CI system. As mentioned, some tests may have to run on a lot of different hardware combinations in order to cover all the supported areas of an application. There are a lot of different CPU's, GPU's, software, OS's etc which work in different ways, but to generalize this project in order to be representable for several use cases, it's important to not focus too much on the details and select dimensions which are common for most or all CI systems. It can also be tempting to select dimensions that are easy to implement, but as each hardware resource comes with different characteristics which influence how the system will work, it is important to focus on the most critical resources.

4.3.1 Software as a dimension

Let's look into how selecting different resources as dimensions will affect the solution. Network, CPU, memory, disk and software are the 5 bottlenecks in computing identified by Schultz (Schultz, 2017). If software would be selected as a dimension, it's either

there, or it's not. It would be hard to turn software into a scale. If a computer has some of the software that is required to run a job, but not all, it can't run there, leaving it to be a very simple dimension to use. Software is also a resource you can't run out of, except in rare cases where number of processes are limited by licences. The question is then, how relevant is it to use it as a dimension? If some software is required for most of the jobs, it's natural to have it installed on all the agents. It is also common to build docker containers within a job so that all the required software is installed within the container and doesn't have to be on the agent itself. The only truly important software to have on an agents then is docker, which in that case should be installed on all the hosts. This setup removes the requirement to check for software on agents, as all required software is installed in the setup step of a job.

4.3.2 Disk space as a dimension

Disk space can also affect a continuous integration system based on the setup. Again, if the jobs are set up as docker containers, it wont affect how the CI system runs by much. Docker containers are by default setup to use maximum 10GB, which isn't a lot, though it can be changed. It is also natural to remove containers after a job is finished to make sure they don't stay on the system and use up space without being in use. If logs are gathered and stored after a container is finished, it also seems unnatural to store it on the agents. It is more common to send and gather logs in a common logging system and plenty of log management systems exist for this purpose. If space should become an issue, it can be selected as a dimension, but for a CI system, a cleanup after a job or a cleanup on a regular basis will in most cases cover it. If an agent runs out of space however, the current job will return errors and the pipeline will fail, which sends a clear signal that a cleanup is needed.

Disk I/O speed is a different aspect that could be a bottleneck in a system. I/O stands for input/output operations to a disk, commonly known as reading and writing to disk. When a job starts in a CI system, a work space folder is created and the repository code is cloned to the directory. This action alone can be slow if the I/O disk speed is slow.

Therefore, it's important to inspect I/O performance to increase job speed, and placing jobs on the hosts with faster I/O speed can improve the overall workflow speed. Nevertheless, the job run time can be predicted as long as the job runs on the host alone. Multiple simultaneous jobs on the same host, however, will make the time unpredictable as several jobs may want to use the I/O capacity at the same time. Figure 4.5 shows a screenshot of a Github Action workflow job. The time it takes to setup the job and check out the code is only a few seconds even though the repository that is cloned is of a significant size. No matter if the host has high I/O speed, it still won't be a significant increase in time since a slower speed such as a few seconds is not a lot in relation to the run time of the other steps.

Figure 4.5: Screenshot from a Github Action workflow with a few steps including the time each step takes. When the code checkout and job setup only takes a few seconds, it indicates that the I/O speed is a very small factor of the complete job run-time.

4.3.3 Memory as a dimension

How much memory is available for a job will affect job performance, but only to a point. Conversely, too little memory can more likely cause crashes. The RAM (Random Access Memory) sets the hard upper limit to how many tasks can be performed at the same time as data is stored here for every running operation. An operating system, for instance, often pretend to have more memory available for each process until the process tries to allocate too much memory, and the process is killed in response. How a memory limit will affect the job, however, depends on where the limit is set. If a process runs out of available memory, the process can freeze or even cause an out-of-memory exception which will cause a job to fail. If the job asks for more memory than whats available on

the system, the job is killed, also causing the job to fail. Increasing the memory above this will increase the speed of the process up to a limit where increasing the memory won't show any significant differences. The CPU, which will be addressed in the next paragraph, and RAM are however closely working together and can limit each other. Combining one bad and one good resource can be seen as letting a fast employee work with a slow machine or the other way around. It is therefore important to explore which one is running at full capacity when processes are running slow to know which of the resources is the bottleneck for a single job. The Linux operation system also has the ability to allocate part of a disk as swap space. This space can then be used as additional memory if needed, it does however require the host machine to be set up with a swap space in addition to the regular disks.

4.3.4 CPU as a dimension

The CPU is the brain of a computer. Using CPU as a dimension for bin-packing will, like memory, affect how fast a job can finish. CPUs have two dimensions in itself, the number of cores, but also the clock speed of the cores. The number of cores decides how many tasks can run simultaneously while the clock speed determines how fast a single task will finish. Deciding on which one is the most important to increase can be difficult as they both play an important part in increasing efficiency. Finishing tasks quickly opens room to perform more tasks, and running more tasks simultaneously does the same. As CloudBees recommended, the number of executors per CPU core should maximum be $nExecutors = nCores - 1$, other than that the requirements will vary depending on what each job requires (CloudBees, 2022). This means that at least one CPU core is needed for each task, but there is also an upper plateau where using more cores for a job isn't improving the completion time as there is a limit to how many parallel computer tasks are performed by a job. The results therefore depend on what a job actually does. A python script, for instance, only uses a single core, though there are scientific libraries such as numpy and multiprocessing libraries such as the multiprocessing module that have the ability to utilize several or all of the cores at the same time. Being able to detect the requirements and usage of each job is therefore

important to be able to place them optimally across the available agents.

4.3.5 Network as a dimension

As Bhardwaj mentioned, CPU and memory bottlenecks can be the main sources of a slow performing CI (Bhardwaj, 2021). Using CPU and memory as a dimension for bin-packing, therefore, seems inevitable to be able to perform better and more efficiently, which is the goal. Nevertheless, there are other dimensions that also affects the efficiency of job completion. Network or bandwidth can affect installation of job requirements, communication between agents and the controller and the upload speed of artifacts, but there has been no discovery of network being mentioned as a bottleneck for CI systems. Bandwidth is in general very important for software developers, and the installation of high performance network equipment in general can eliminate this as a bottleneck from a CI system.

4.3.6 Assessing the suitability of a dimension over time

In their article, Kumaraswamy and Nair based their results on number of servers required and CPU utilization (Kumaraswamy & Nair, 2019). What should be the main focus, however, should be based on the bottleneck of the specific system. If the simulation implemented to evaluate bin-packing algorithms both returns the number of bins required together with how well each resource is utilized in a bin, it’s easy to see where there is improvement potential and where the main bottlenecks are. It can also reveal the best area for potential improvement, for instance if a CPU should be upgraded or if a RAM card can be moved to a different host. The amount of resources allocated to a single job should also be evaluated after observing a job over time. Starting by giving a job a high amount of resources and from there reducing it little by little to see how it affect the job will eventually give a reasonable amount of information to decide where the limit should be set. The main evaluation on which dimensions to focus on, however, can be based on how many jobs can run simultaneously, how well all resources are utilized or on how quickly a pipeline is able to finish.

4.3.7 The difficulties of assigning resources

The jobs, as mentioned, have to be assigned resources, either by setting a specific amount or an upper or lower limit. This is an approach that fits well into an existing Kubernetes cluster setup. It is, however, hard to determine how much minimum resources each job should have or what its maximum use might be. I propose that the best way to determine where the limits should be set is to observe the jobs over time. If the jobs are assigned very little resources to begin with and gradually increase the amount, it can be observed where the plateau is, and that can be an option to use as a lower limit. Where the job run time doesn't increase with increasing resources is a good starting point to set the upper resource limit. Observing a host's resource utilization over time is also a good way to discover if a job's resource limits should be increased or decreased. Determining job resource limits is time consuming, hard to do and not a part of bin-packing at all. Implementing job resource limits can therefore be hard, especially for new jobs and pipelines where no resource usage is recorded over time.

4.3.8 The path to follow and a path left unexplored

Selecting dimensions will affect how many dimensions are used for the bin-packing algorithm. The more dimensions, the more evaluations has to be made for each item placement. Some dimensions could be considered more important as they are the main bottleneck of a system, some dimensions may require setting a hard limit while some can be overstepped as it will only slow a process down. One aspect that will be discussed in more details later on is the fact that job sizes vary, which also opens more room to set the resource limits in different ways. If the job resource limits are set too low, the CI jobs will slow down or, in worst case, crash. If, on the other hand, the resource limits are set too high, there will be a lot of unused resources, therefore not fulfilling the goal of this assignment of optimizing the resource utilization.

There are a lot of potential dimensions to choose from and it can, therefore, be hard to select the correct ones that will cover most use cases. Different companies also have

different requirements and the dimensions for their CI should then be evaluated for their individual needs. There might exist dimensions not identified and discussed in this paper that can be bottlenecks and extremely important to a company, which can make combining bin-packing and CI software testing impossible. This is, however, considered an edge case. Nevertheless, selecting the wrong resources for this assignment can cause the results to not correspond with reality and therefore not correctly addressing the problem from a real world perspective. The safest dimension to approach to not exclude many use cases is focusing on CPU, which has, as discussed, been used in other IT approaches using bin-packing for instance by Kumaraswamy and Nair (Kumaraswamy & Nair, 2019). The job completion speed does, however, also require enough memory to be able to fully utilize the CPU and, therefore this is selected as the second dimension. These are also the two main Kuberenetes resources and adding additional custom resources is therefore not necessary.

CPU and memory has been selected as the path we follow going forward with this project. Other dimensions will be left behind at this point and will therefore not be explored further. Network speed, installed software and disk space are some paths that will not be explored further as well as some other potential bottlenecks such as GPU and port collision. These are dimensions that can be discussed more in detail in other research built on top of this assignment.

4.4 Bin-packing in software testing

A factor in software testing that differs from bin-packing, is that once an item is placed, it can’t be relocated. It has to finish the whole job before it will be removed. In a physical bin, items can be removed and used to fill a bin more optimally if the original placement isn’t good enough, though this isn’t desired and also something most bin-packing algorithms don’t take into account.

Another interesting aspect that is often the case for software testing, but isn’t very common in bin-packing solutions, is that the bin-sizes can vary. The order of the bins

may therefore affect the bin-packing result as well as the placing algorithm. Optimal job scheduling is an approach that has a separate approach category which focuses on this, called the uniform machine scheduling. The approach is often to place items on the machines where the job can finish the fastest, and this depends on the available resources in the individual computer. So while a bin-packing approach doesn't focus on the differences of the bin, a job scheduling approach can be used to select the bin where the job will finish faster.

In software testing, an item's resource requirements may also vary over time. As mentioned, a job consist of several steps where some steps may require different amounts of resources, and the setup itself may be the most resource demanding depending on what is being tested. The setup step often consist of building a docker container, cloning the whole repository and downloading and installing software. When simple, smaller tests are ran after this, the resources the test itself requires is less than the actual setup. This is, however, seldom the case when running stress-testing, recursive testing or similar. Then the test itself will try to use as much resources as possible to either test the capacity of the system or use a lot of resources to complete multiple actions simultaneously. Figure 4.6 shows a graph of three separate processes, where the CPU usage varies over time. The blue graph has high CPU usage on setup, the green has increasing CPU usage over time while the red process remain relatively stable. Nevertheless, taking the variation into account while placing the item can be very important, yet challenging. Should the size of the item be set based on the maximum resource consumption, the least, the current usage at the evaluation time or something in between? This will also affect how to place the object optimally.

When using on-premise hardware, the hardware is seldom turned on and off depending on incoming workflows. Karthik et al., on the other hand, has a green focus while using bin-packing in data centers with the goal of reducing power consumption (Karthik et al., 2016). Not turning the machines off to save power sets power consumption to a constant and removes the need to take it into account when placing

Figure 4.6: A graph showing the CPU usage of three separate processes. (Hwang et al., 2016).

items. The problem can therefore be investigated from a different perspective: How can we optimally place items to *best* fill the available bins, as opposed to an original bin-packing approach which has the goal of using the fewest possible bins. Since resource usage varies over time, giving a job more resources could reduce the amount of time a single job needs to complete. It might therefore be interesting to investigate a different initial approach of placing items into bins: start by placing one item in each bin. If the setup is the most resource demanding in the bin-packing process, opening all the bins and placing items evenly is definitely an approach that should be considered. Placing one item in one bin each to begin with can ensure that the tasks finish faster so that more resources are available when the next items are being placed. Figure 4.7 shows how regular bin-packing would optimally place items to reduce the number of bins used while figure 4.8 shows an example where the items are spread out more equally, therefore leaving more resources available that the jobs can use in addition if needed. All the machines are utilized in figure 4.8 instead of leaving one or more computers idle in figure 4.7 while the jobs are fighting for resources on the other machines.

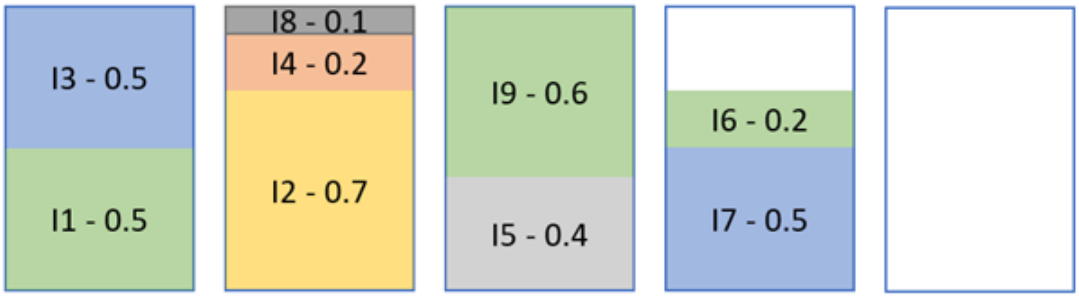

Figure 4.7: Illustration of an example of an optimal packing solution in regular bin packing, the rightmost server is empty and could be used to run tests

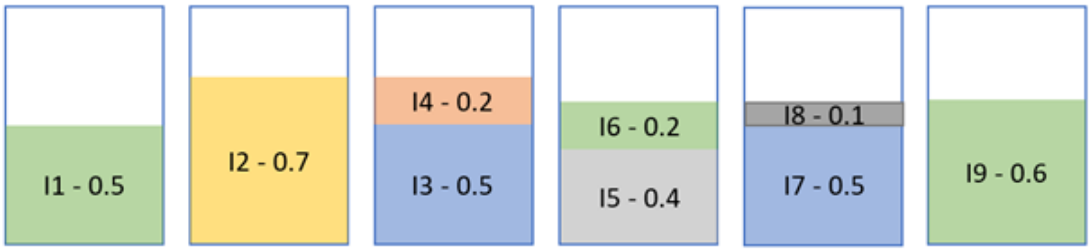

Figure 4.8: Illustration of items spread more equally across the available bins. This can be seen as a more optimal approach for bin-packing in software testing as there are more resources available to the items.

4.5 An alternate perspective: Optimal Job scheduling

Optimal job scheduling is an approach that should be considered and tested in software testing because it considers where a job can run the fastest. In the background chapter it is argued that single-stage job scheduling is the approach that is most fitting in a software testing area. The four types of single-stage job scheduling are: Single-machine scheduling, identical-machine scheduling, uniform-machine scheduling and unrelated-machine scheduling. The single-machine scheduling isn't relevant in this case unless there is only one host available. Identical-machine scheduling can be relevant if the hosts are all the same, but the most interesting one is the uniform-machine scheduling which uses multiple different hosts. This specific approach solves the problem in software testing that bin-packing rarely focuses on: that the bin sizes may vary.

In job scheduling there are, however, some constraints that are not relevant for a CI job scheduling solution. A deadline to finish all jobs cannot be used if developers constantly trigger new pipelines. The solution also cannot be measured by when all jobs finish unless a limited input stream is used for testing. The job scheduling approach usually doesn't allow more jobs to run simultaneously on one machine either and the available resources to a job is therefore constant. The time required to finish a task in this case doesn't vary based on how much of the resources are used for other tasks. One thing that can be done to make the job scheduling approach more representable is to set the upper and lower resource limits for each job within a range that makes sure that the task completion time doesn't change based on how much resources are available. The lower and upper limit could for instance be the same. The downside is that this approach doesn't take the variation of resource requirements into account and there will be unused resources. This approach alone therefore also has downsides and while it may improve the situation it doesn't solve all the problems software testing faces when trying to optimize resource usage.

4.6 Optimizing item placement using a combination of two bin-packing algorithms

Using a single bin-packing algorithm for optimal placement, doesn't seem to completely fulfil the criteria of resource optimization in a continuous integration system. This is due to the various differences from software testing, together with the fact that the optimal placement will vary depending on the current queue state. There are two main queue states that will affect how items should be placed:

- 1. Available bin capacity, no item queue
- 2. Full bins, long item queue

The first state will be in a situation where every new item in the queue can be placed immediately, while the second has several items that need placement and they can therefore be placed in a different order if this would optimize the overall results. These are in most cases independent states, unless a lot of items are placed in the queue in a short period, as the system might not be able to place the items quickly enough to empty the queue immediately. The other exception is if the bins are already full, but no new items are added to the queue for a little while, and the bins will slowly finish their tasks. Full bins and a queue will again leaves us with two options, a locked queue where the first object in the queue has to be picked before the next, or an open queue where any item in the queue can be picked at any time.

With a locked queue, the item to be placed is always decided for us, the algorithm therefore only has to select the bin to place the item in. In bin-packing terms this equals an offline queue, where the items are known, however, they won't be sorted as offline bin packing approaches usually do. An open queue, on the other hand, opens up many new placement opportunities. If the goal is to finish as many jobs as quickly as possible, the hosts are filled up with small jobs and the bigger jobs will continuously be pushed backwards in the queue. In this case a full pipeline with a mix of job sizes will take a long time to finish even though a lot of jobs finish quickly. Sorting the other way around and

placing the bigger items first may leave room to place smaller items in between, but the issue of jobs continuously being placed towards the back of the queue will still be an issue.

To solve the queue sorting issue, a new variable should be introduced: a maximum queue time. When an item has been in the queue for a certain amount of time, it has to be placed at the next opportunity. This is harder to do when the smaller items are placed first as it is likely that several items have to finish in a single bin before the bigger item can be placed and a lot of room can be available in other bins to continue placement there. If the bigger items are placed earlier on the other hand, queue sorting seems more optimal.

A second approach using an open queue is to have the goal of filling all bins. This means placing items whenever there is room for them no matter the order of the queue. One problem with this approach is that smaller items can fill up small spaces easily while bigger items can't. The theory is that eventually all bins will only contain smaller items and the bigger items will never fit. New small items will replace small items every time and bigger items will queue indefinitely.

One thing that should be strongly considered while selecting the order of the items is that there are several other factors that will affect how efficiently jobs finish. Some jobs depend on other jobs, and once a job finishes a lot of others may enter the queue. Jobs with no jobs depending on them are therefore less important to run early because the pipeline doesn't finish before everything else finishes anyways. Jobs that fail often could also be placed early to be able to cancel the workflow as early as possible if it is going to fail. Taking the downsides to sorting the queue into account, the items for this approach will be in a locked queue.

When the goal is to remove queue time completely, the first queue state, when there is bin space available and no queue, is the one that should be considered. Nevertheless,

with lack of resources, the second queue state is more likely to occur and the goal will be to reduce queue time. The approach should therefore consider both of these states and the best approach is therefore hard to pin down to a single algorithm. The following approach is therefore proposed:

- For queue state 1, when a job is placed in an empty queue and there is room to place the item immediately, the worst-fit bin-packing algorithm should be used. This will cause all the items to be placed out evenly and all machines will be utilized, leaving more resources available for the jobs to be able to finish faster.
- For queue state 2, when a job is added to the queue and it cannot be placed immediately, the placement algorithm should be changed to use the best-fit algorithm to be able to fill the remaining space in the bins leaving each bin as full as possible.

Changing the algorithm based on several criteria is not a common way of using bin-packing. This approach of combining algorithms can be called an adaptive algorithm which changes based on queue state. The trigger to change between the algorithms is checked whenever a new item enters the queue or when an item from the queue is being placed. If there is one job that doesn't fit and the algorithm used at the moment is the worst-fit algorithm, then it should change into the best-fit algorithm. If the current algorithm is the best-fit algorithm and there is no queue and the job can be placed immediately after entering the queue, the algorithm is changed to the worst-fit algorithm. This is the easiest way to know exactly when to switch algorithm and it doesn't require any prior knowledge of patters or expected jobs.

A different trigger condition could be set after observing the CI system over time. A specific time of day when developers arrive at work and go home from work could also be the trigger point for changing between the algorithms in order to be prepared for an expected situation. If there are certain other conditions that often lead to a switch in queue state, the algorithm can change a short period of time before the queue is filled up to be better prepared for the expected queue state. This, however, requires learning

over time and discovering when or what usually happens before the queue state is changed. The first approach is therefore easier to use, especially before any data is recorded about typical queue behavior. Nevertheless, the disadvantage to not changing ahead of expected queue states will cause the first items that should be placed using the best-fit algorithm to wait for enough space to clear up in a bin, when starting the best-fit algorithm earlier could have made sure enough space was left to at least place a few more items.

One aspect that this approach doesn't consider is that some machines are faster than others. Some job scheduling approaches place items on a machine that will ensure that the item finishes as fast as possible, thus, placing the items on the fastest machines. This can be added to this approach by sorting the bins or host machines so that the first bin evaluated for placement always is the fastest. Let's call the combined approach, the Resource-Optimized-Software-Testing algorithm or ROST algorithm. Below is a short code sample of how to sort the bins based on CPU clock-speed before running the algorithm.

```
1 all_bins_sorted = all_bins.sort(key=lambda bin: bin.clock_speed,
    reverse=True)
```

4.7 Defining the case scenario

To test the ROST algorithm through a simulation, the first step is to create a case scenario. The case scenario might not be completely realistic as there are many factors that play a part, but the assumptions are based on observations from real situations. The simulation is a tool to try to compare the model created with other possible solutions to establish the usefulness of the selected path. Since the case scenario might be unrealistic, it can't be compared to other studies, and the same case scenario variables created here therefore have to be tested with several approaches in order to be truly comparable.

Selecting how a typical job and bin look is the first step. Starting with the bins, it's important to recreate a typical hardware setup. To create a simple case scenario which don't require too many jobs, 4 bins are used. As these on-premise resources are supposed to handle quite some load, it is natural to select components from the higher end, therefore typically go for CPUs with at least 6 cores, mostly 8 cores and maybe even use a 12 core CPU. Based on Horowitz' proposition in the background chapter (Horowitz, 2020) the following four computer CPUs are suggested: [6, 8, 8, 12]. The speed of each CPU core should also be decided as this will define how quickly a job can finish. As this doesn't decide if an item fits in a bin or not, this is not a dimension in a bin, but can decide the order the bins are sorted. Based on the list of best-selling CPU's on Amazon (ArsTech, 2020), the CPU speeds are set to [4.2, 4.4, 4.1, 4.6]. Next, the memory has to be decided for the above computers. It's preferable to use higher amounts of RAM together with the better CPU's. Based on Kingston' proposition mentioned in the background chapter (Kingston, 2019) the following memory to the four hosts is suggested: [8, 16 16, 32]. To make the simulation easier, the speed of the memory is all assumed to be the same.

In addition, the job requirements have to be decided. The total number of jobs need to be able to at least fill all the bins and a little bit more. It can be hard to define the size of a job, specifically when the resource usage varies over time with the setup step and the actual testing. By using the assumption that the smallest computer can be hard-coded to fit two jobs, the smallest computer has to be able to fit minimum the two biggest jobs in the workflow at the same time or the biggest job twice. In a perfect world, the biggest job will at maximum size perfectly fit twice into the smallest host, we're giving it the requirements of 3 CPU cores and 4 GB of RAM at max usage.Therefore, all jobs should be created randomly within the range of 1 too 3 CPUs or 1 to 4 GB of memory, to cover many different test cases. The resource usage will also be decreased after the setup stage. The resources required after the setup is something that also has to be observed over time in an actual system, but for now, to illustrate the concept, one CPU core and one GB of RAM is removed from the requirements, if possible, after the setup is

complete.

The number of cores and memory required often also reflects the size of the job and the run-time for the individual job should be set based on these. For now, the individual job run-time is set to $(CPU + RAM) * factor$ seconds where the factor is used to increase run-time if we want to observe the system over time. To begin with, the factor is set to 10 for the individual job run-time, but the factor should be a parameter so it's easy to adjust. The factor is added to ensure that the jobs at least are big enough to run for a few seconds when the run-time on the individual machines is divided with the CPU clock-speed. We assume that the setup time is complete halfway in the run-time.

4.8 Evaluating the simulation

When creating and running the simulation, it's also important to know how the results should be evaluated. This is an important part of creating a model, but it can be hard to evaluate results from an exploratory assignments as the goal isn't strictly defined from the beginning. Nevertheless, there are several propositions that have been discussed as important and some that have been argued to be less important.

One thing that has been considered less important is the time an algorithm takes. The portion of time the algorithm uses is very small compared to actually running jobs. The algorithm execution time will also be predictable and the model should therefore not be evaluated based on how much time they take. Nevertheless, the complete run-time is important as improving the resource utilization in the end is done to increase efficiency. The main unpredictable time estimation is the queue time, but also the complete workflow run-time as the speed of the different hosts vary. The most important evaluation, therefore, comes down to how fast a large set of jobs can finish compared to other methods. This evaluation isn't something used in bin-packing, but in job-scheduling and it is more fitting with the software testing concept.

Another obvious way of evaluating a system is how many jobs can run simultaneously. While bin-packing often measures the results of the algorithm based on how few bins a finite number of items require, fitting as many items into a finite number of bins is a more fitting approach for software testing as the number jobs most commonly is unknown and can be more than what the bins can fit. The number of items fit into each bin is also a similar measurement to ensure clear, numerical and comparable results. The problem is that the input flow and job variation can be random and therefore cause differences between each run. It is therefore important to test with different input flows and compare it with a different approach using the same input flows. It is, however, harder to evaluate based on this while the simulation is running, as jobs continuously move in and out of the system.

The utilization of each individual resource is also a way of measuring the approaches efficiency. The resource utilization is a common evaluation used in bin-packing. Nevertheless, the same problems will be faced with this measurement as with the number of jobs as the bins continuously change and the input workflow has a huge impact on the results. It is interesting to investigate how the resource utilization along the way affect the complete workflow run-time, and a continuous print can show the situation to the user, but logging it in a good, comparable way is a different challenge.

Finally, the conclusion on how to evaluate the results of the simulation will be the complete run-time of the set of jobs generated. The same job input flow will be used in all the approaches to give comparable results, and several input flows will be used to ensure that specific input flows wont affect the results. The bin utilization will be printed continuously to the user during the placement to be able to use the data for further investigation and discussion of the results.

4.9 Exploring the possibility for implementation

In the end, the approach is only usable if it is actually implementable. The next step is therefore to explore what is required to implement a combination of bin-packing with a CI

system.

Most CI systems, as mentioned, uses a container orchestration service for auto-scaling which makes it possible to create new runners/agents on the hosts and delete them after a job is done. Secondly, Kubernetes already supports resource managed scaling where a runner and host can be assigned resources based on the job's requirements. The algorithm used to assign runners to hosts, however, isn't specified, but it's possible to change the Kubernetes scheduler which decides the algorithm used. With all the groundwork already existing, the implementation seems doable.

Adjusting the individual job requirements, however, is a challenge. The queue of jobs in the CI systems will send requests to create new runners. In Kubernetes a job resource usage is set based on the request which is the minimum amount the resource needs to run and a limit which is the highest amount it should be able to use. The scheduler uses the request and limit to optimally place the newly created runner on a host. Deciding on the request and limit of a job's resources, however, require logging and calculations of resource usage over time. The simplest implementation from the CI providers would be for the developers to set the resources manually when creating a new job, but this is also something that can be done using artificial intelligence. If the system can set default values for a job and adjust them over time, this tool would be a lot easier to use for the developers. This does, however, require some work from the CI providers.

The blog written by Madel stated that auto-scaling and resource quotas don't mix (Madel, 2018). The reason behind this was that the solution is mainly created to create new hosts in a cloud cluster and when adding a new host, the resources has to be set manually. When the solution, however, is used to scale runners on existing hosts, all the hosts' resources are already known. Adding a new host, however, will require some manual work. Nevertheless, the scaling in this case doesn't adjust hosts, but runners on the hosts and no manual work is required while auto-scaling.

The question is then, why hasn't this been done before? The simulation will show us how valuable combining bin-packing with software testing is and give us an answer to if this approach is useful. If it shows that the approach actually utilizes resources better and improves the speed of running workflows, the next challenge is the implementation. Without artificial intelligence, to set job resource requirements is a huge manual workload for the users of the CI. On the other hand, implementing an artificial intelligence approach demands a lot of work from the CI providers. There might be many other things within a CI system that is more requested by the users and is less resource costly for the CI providers to implement, which therefore has a higher priority. Creating value for the users or customers are often a priority, and this might have a too high cost value compared to user value to spend time on. This approach also only targets the users which has on-premises hosts and doesn't provide cloud users any advantages, which again lower the usefulness of this approach. Staying ahead of the competitors, however, is important in a large pool of CI providers, and this could give a competitive advantage. To come to a final conclusion on the question, the results of the simulation has to be analyzed.

4.10 Analysis

How well does actually software testing and bin-packing work together, and why haven't these two concepts been combined earlier? A model has been created, using the bin-packing concept within the restrictions and needs of a continuous integration system. The approach that was discussed describes a solution in which not one, but two algorithms are used to fill the gaps bin-packing is lacking by itself. In the end, a concept from job scheduling is also used to properly be able to utilize all the available resources in the best way.

Two dimensions were selected as the most important to focus on while placing CI jobs onto hosts. There is, however, a lot of room to make mistakes while selecting dimensions and the needs of individual companies vary, making it difficult to select dimensions that will fit all. This approach, however, can be adapted to fit with different

dimensions if necessary, but for the simulation, CPU and memory will be used.

There are also more algorithms that haven't been fully evaluated or even mentioned in this paper. As well as other algorithms and other dimensions, there are also other approaches to reducing CI queue time. This area has not nearly been explored enough. The simplest, but also most expensive solution is to buy more computers. A brute-force solution like that, would not be sustainable economically nor environmentally.

If we for a moment look away from the model we have created and look at the bin-packing approach as a whole, the main goal is to avoid unwanted situations. One of these situations is starvation of jobs where some jobs never get to actually be placed, which is caused by sorting a queue continuously when the hosts are full. Offline bin-packing has the advantage of knowing the items in the queue and can sort the items based on what will optimally fill the bins, but in a software testing perspective, the items which ends up in the back of the queue can be starved. With infinite bins and few jobs, sorting can help place items optimally, without the problem of starvation, but then again, there wont be a space issue to begin with. Sorting items is therefore not optimal within a continuous integration system.

Another unwanted situation in bin-packing is that bins aren't properly filled and a lot of space remains. Using the worst-fit algorithm can often cause this scenario. When all bins are filled using the worst-fit algorithm, the next item can't be placed immediately. A switch to the best-fit algorithm earlier will ensure fuller bins and more items get placed. The problem, however, is that it can be difficult to find a fitting trigger condition to switch at the right time. One suggestion can for example be to switch when some or all bins have reached a threshold, but setting that limit can be hard and the variation of items and bins will cause a lot of different placement situations that will affect when the trigger condition actually kicks in. When the trigger condition is set to change from worst-fit to best-fit when the next item doesn't fit, the possibility of left-over space is high as the purpose of the worst-fit algorithm is exactly that, to put the item in the bin which leaves

the most remaining space.

These unwanted situations come from concepts within bin-packing that don't properly fit into a software testing solution. This isn't a match made in heaven where everything naturally fits together without adjustments. At first glance, bin-packing seemed to solve most of the CI resource utilization issues, but when looking at the details, there are several unwanted situations and problems with combining the two concepts.

4.10.1 The simulation

To evaluate the model, the ROST algorithm, it is interesting to see how it works compared to other solutions. A python script is used to simulate the solution and compare it to other approaches. The worst fit, best fit, optimal job scheduling approach is run individually as well as an approach where two items are placed in each bin no matter the space. The results are then compared to the results of the ROST algorithm.

4.10.2 Creating jobs

In the Python simulation, jobs and items are created and then used to run each of the approaches. A job has the variables CPU, RAM and run-time. The run-time is set based on the CPU and RAM requirements multiplied by a factor to increase the run-time so the results will be measurable. In addition, each job has a number to keep track of the items.

```
70 class _Job():
71     def __init__(self, cpu, ram, num, job_length_factor):
72         self.cpu = cpu
73         self.ram = ram
74         self.num = num
75         self.runtime = (cpu+ram)*job_length_factor
76 
77     def __str__(self):
```

```
78          return f"Item {self.num} has requirements: cpu:{self.cpu
              }, ram:{self.ram}"
```

To cover several situations, the items are random generated for each run. This is done by setting the lower limit of the CPU and RAM to 1 and the upper limit for CPU to 3 and the upper limit for RAM to 4. These limits ensure that at least two items of the biggest size fits in the smallest bin. The same items are used for all approaches within a single run, but are recreated when re-running the full simulation to investigate how different items affect the results.

```
87 def _create_tests(job_length_factor = 10) -> None: # Random
     generate 20 jobs within the given interval values
88     for i in range(20):
89         _BinVariables.bin_items.append(
90             _Job(cpu = random.randint(1,3), ram = random.randint
                  (1,4), num=i, job_length_factor=job_length_factor)
91         )
```

4.10.3 Creating bins

A bin has the parameters CPU, RAM and clock-speed, as well as an identifying number. When the object is created, an empty list is also created to be able to put items in it later on. The CPU and RAM is saved to two variables when the object is created to be able to edit one of them as the items go in and out while keeping track of the overall capacity. The bin object also have a few methods, the __str__ method is overridden to print how a bin and the bin content look at the print time. This is to be able to observe how the bins fill during the simulation. Figure 4.9 shows how a single bin is printed where the width represent the CPU and the height represent the RAM.

The bin object also has a method for placing items and one for removing the item. When the item is placed, the item is added to the bin's item list and the available CPU and RAM for the bin is reduced. A thread is also created which runs the method for

Figure 4.9: A screenshot from the simulation showing the print of a single bin. The width represent the CPU and the height represent the RAM.

removing the item. This method sets the timer until the item should be removed from the bin. Halfway through the time, the resource usage of the item is reduced by one, if it uses more than one already, and when the rest of the time has passed, the full item is removed from the bin. The complete code for the bin can be seen below:

```
11 class _Bin():
12     def __init__(self, cpu, ram, num, clock_speed):
13         self.full_cpu_capacity = cpu # The full capacity is set
               as well as one variable that will vary based on item
               content
14         self.full_ram_capacity = ram
15         self.cpu = cpu
16         self.ram = ram
17         self.num = num
18         self.items = []
19         self.clock_speed = clock_speed
20
21
22     def __repr__(self):
23         return str(self)
24
25
```

```
26      def __str__(self): # Draw bin showing capacity
27          bin_drawing = ("\n" + " " + "_" * self.full_cpu_capacity
              + "\n")
28          for i in range(self.full_ram_capacity):
29              if i == self.ram:
30                  bin_drawing += "|" + "¯" * (self.
                      full_cpu_capacity - self.cpu) + "|" + " " * (
                      self.cpu-1) + "|" + "\n"
31              elif i > self.ram:
32                  bin_drawing += "|" + " " * (self.
                      full_cpu_capacity - self.cpu)+ "|" + " " * (
                      self.cpu-1) + "|" + "\n"
33              else:
34                  bin_drawing += "|" + " " * (self.
                      full_cpu_capacity) + "|" + "\n"
35          bin_drawing += (" " + "¯" * self.full_cpu_capacity + "\n"
              )
36          return bin_drawing
37
38
39      def remove_item(self, item):
40          runtime = item.runtime/self.clock_speed
41          time.sleep(runtime/2) # after half the time reduce
              resource usage
42          cpu_reduced = False
43          ram_reduced = False
44          if item.cpu > 1:
45              cpu_reduced = True
46              self.cpu += 1
47          if item.ram > 1:
```

```
48              ram_reduced = True
49              self.ram += 1
50          time.sleep(runtime/2)
51          self.items.remove(item)
52          if cpu_reduced:
53              self.cpu += (item.cpu-1)
54          else:
55              self.cpu += item.cpu
56          if ram_reduced:
57              self.ram += (item.ram-1)
58          else:
59              self.ram += item.ram
60
61
62      def place_item(self, item):
63          self.items.append(item)
64          self.cpu -= item.cpu
65          self.ram -= item.ram
66          countdown = threading.Thread(target=self.remove_item,
              args=(item,))
67          countdown.start()
```

4.10.4 The algorithms tested

After the items and bins are created, the algorithms can be run. There are five different approaches being tested. The first one is the hard-coded approach, where no matter the available bin-space, the bin can hold two items. The second and third approach are common bin-packing approaches, the worst-fit and best-fit approach. The fourth approach is a combination of job-scheduling and a first-fit approach, the bins are sorted by clock-speed in decreasing order and then a first-fit bin-packing approach is used to

place items on the first host available. The final approach is the ROST algorithm approach, which starts with a job-scheduling approach by placing one item in each bin, where the bins are sorted by clock-speed in decreasing order. Thereafter, the items are placed using either worst-fit or best-fit based on the current queue state. It starts with the worst-fit approach until the next item can't fit in any bin, then the algorithm is changed to the best-fit. If the next item, however, fits into more than half the bins, the algorithm is switched back to the worst-fit approach. This continues until all items are placed. The implementation of each algorithm can be seen in the Appendix 7.1.

4.10.5 The output of the simulation

During the simulation, the current bin situation can be printed in a table to easily observe how the process is going.

```
258 def _create_pretty_table() -> None:
259     os.system('clear') # cls on Windows, clear on unix
260     table = PrettyTable()
261     table.field_names = ["Agent 1", "Agent 2", "Agent 3", "Agent
        4"]
262     agents = _BinVariables.available_bins
263     table.add_row([str(agents[0]), str(agents[1]), str(agents[2])
        , str(agents[3])])
264     print(table)
```

Figure 4.10 is a screenshot of the terminal during execution of the simulation. The immediate observation is that the CPU seems to be the bottle neck in all the bins, but especially in the larger ones, with the current setup of bins and items. This can be due to several factors:

- 1. The bins have too much RAM in relation to CPU
- 2. The items should have higher RAM requirements in relation to CPU
- 3. CPU is more commonly a bottleneck in hosts

The first two factors can be caused by selecting wrong CPU and RAM for the hosts and items in this project or this can be how typical setups and items look, which therefore leads to the third situation, that CPU is a more common bottleneck in a system.

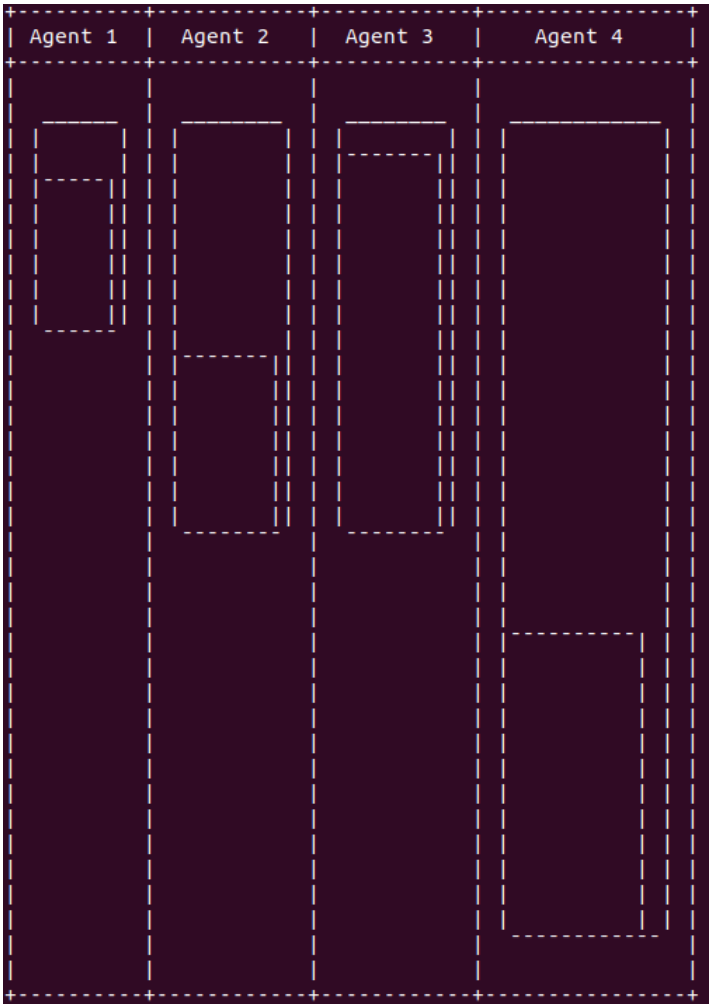

Figure 4.10: A screenshot from the simulation showing a table containing the four bins named Agent 1-4. The width of each of the four bins represent the CPU and the height represent the RAM.

The worst-fit and best-fit algorithm evaluates the best placement based on remaining space of CPU and RAM, but after watching the bins and seeing that CPU is more likely to be a bottleneck, changing the evaluation to purely focus on CPU if the item fits in both dimensions can provide better results.

Another observation made when running the simulation was that in the ROST algorithm approach, the switching of the algorithms happens almost immediately, which means that the first items are placed very fast and the algorithm itself doesn't use much time, which was the assumption. It is therefore running the jobs which uses most of the time. As this is a simulation to efficiently gather data, the job run-times are also set to be much shorter than a normal CI job would be. Each job run-time is only a few seconds compared to what could be several minutes or even hours for particular tests. The speed of the host the job is running on is therefore even more important in a real scenario.

4.10.6 Implementation challenges

There were some challenges with the simulation that didn't occur before implementation, for instance, what to do if all bins are full. The decision was made to wait for one second and then try placing the item again. an other, more complicated approach could be for the bins to send a signal whenever there are changes in the bin and the signal can trigger a new check. this, however, would be more complicated to implement, so the first approach was selected. How to reduce the item size after the setup wasn't completely planned out either. The decision was made to use the initial size for the first half of the run-time, then reduce each resource by one, if they were bigger than one initially. After that, continue for the remaining time before the item is removed.

An additional challenge was to adjust the run-time based on if there was little room left on the host. Setting the run-time based on how many items are in the bin already is a possibility, but changing it after placement if a new item enters the bin is more difficult. The function has to check regularly if anything is added to the bin and from there adjust the remaining time until the job is finished. Due to the complications of implementing this, it was decided not to adjust run-time based on the number of items or the remaining space in the bin. This, however, cause the simulation to stray further from reality and it therefore removes some of the advantages of the worst-fit and the ROST algorithm, while using the worst-fit approach. The worst-fit algorithm was selected due to the fact that it spread the items out, grating them more usable resources while on a host by

themselves, and this won't come to light if there is no penalty to filling one bin at the time.

Another decision made was that no job should be able to use more resources than available on the host so that no bin resources are over-allocated. This restricts the placement of the items, but also doesn't require implementation of resource sharing and adding run-time to all bin items.

4.10.7 Evaluating the results

As previously discussed, the best way of measuring the results from the simulation is to time each of the approaches using the same items and same bins. The algorithms finish when all items are placed and therefore we have to include a function which times the approach as well as the emptying of the bins.

```
245 def _time_approach(function) -> None:
246     start = timer()
247     function()
248     empty_bins = 0
249     while empty_bins != 4:
250         empty_bins = 0
251         for agent in _BinVariables.available_bins:
252             if len(agent.items) == 0:
253                 empty_bins += 1
254     end = timer()
255     return timedelta(seconds=end-start)
```

Since the items are random for each run of the simulation, it was ran ten times to get a good overview of the results. The results can be seen in table 4.1.

The results shows that the ROST algorithm is as fast or faster in seven out of ten runs, is worse in three cases, but on average the results of the ROST algorithm is a little faster

	Hard-coded	Worst-fit	Best-fit	Optimal job-scheduling	ROST algorithm
1	01:17	00:30	00:27	00:30	00:29
2	01:15	00:27	00:29	00:29	00:29
3	01:13	00:33	00:36	00:33	00:32
4	01:14	00:31	00:31	00:32	00:30
5	01:13	00:37	00:33	00:33	00:33
6	01:26	00:38	00:37	00:38	00:37
7	01:15	00:32	00:36	00:32	00:33
8	01:16	00:28	00:26	00:26	00:26
9	01:20	00:33	00:33	00:32	00:31
10	01:14	00:34	00:36	00:33	00:33
Average	01:16	00:32	00:32	00:32	00:31

Table 4.1: The results from the simulation. The simulation is run ten times with the same input for all approaches, but different input for each run. The best run-time for one run of the simulation is in the colored cells. The results are shown in minutes and seconds.

than the other approaches. In two of the three cases where the ROST algorithm isn't the fastest, it comes in second, and in the last case, where it comes in third, it is only a second slower than the two other approaches which shares the first place. All the bin packing and job-scheduling approaches, however, are much better than a hard-coded approach which is typically used for CI systems today. Figure 4.11 shows a graph to easier compare the bin packing approaches to the hard-coded approach and it's easy to see the advantages to using bin-packing for placing CI jobs onto hosts.

Figure 4.12 shows a comparison of the results without the hard-coded approach. The milliseconds from the run-time has been rounded off in table 4.1, but is used in this graph. This shows that while the bin-packing and job-scheduling approaches aren't far from the ROST algorithm, the ROST algorithm is still a little bit more efficient on average.

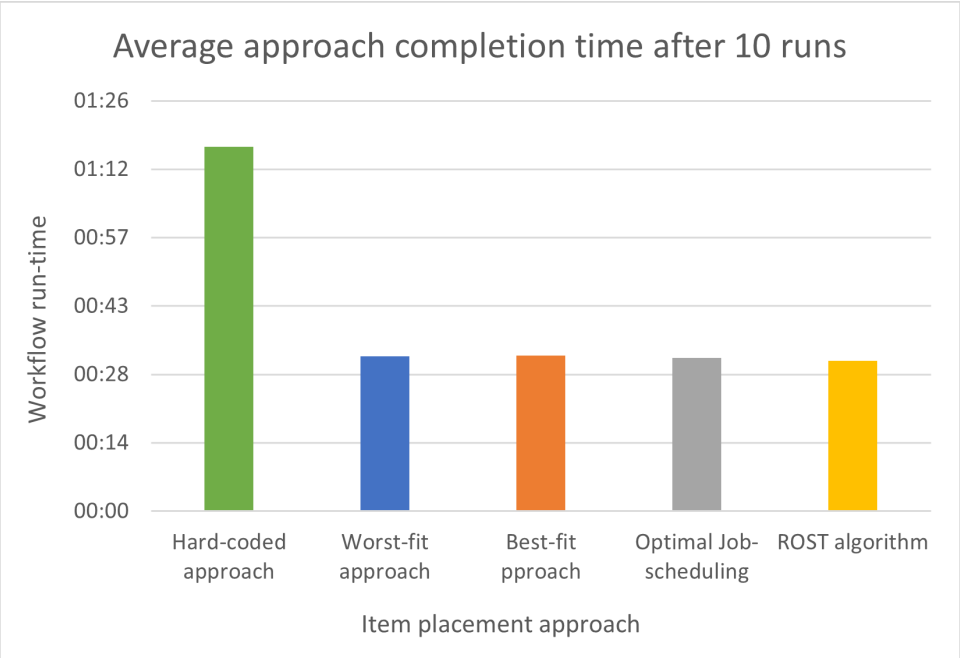

Figure 4.11: A comparison of the average run-time of each approach.

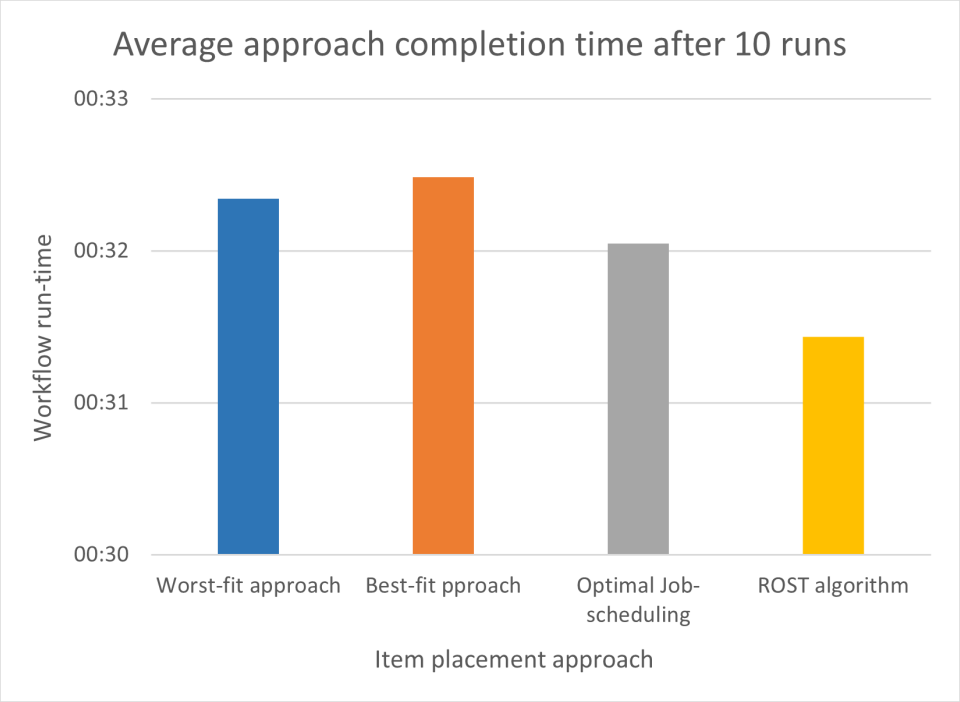

Figure 4.12: A comparison of the average run-time of each approach excluding the hard-coded approach.

Figure 4.13 shows the variation for each run. The differences in run-time is caused by the variation of the job sizes. Each approach uses the same input during the same simulation run, but the jobs are random for each of the ten runs. The graph shows that some inputs grant advantages to some algorithms while other input might be better for other approaches. The worst-fit for instance is the best with the input in run two, while it is the worst with the input in run five. The graph therefore demonstrates how the input strongly affect the effectiveness of the algorithms.

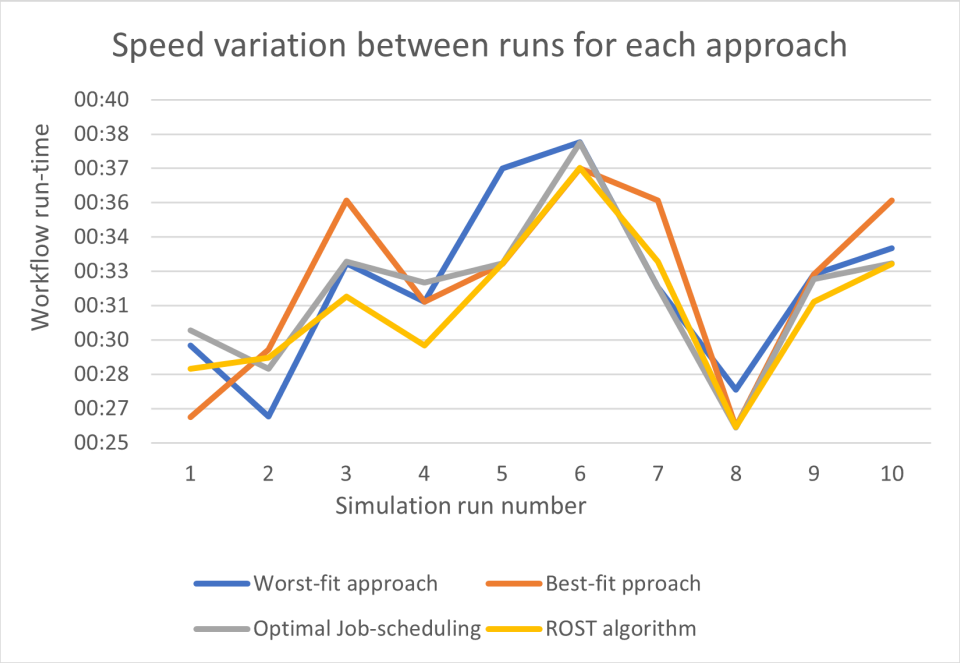

Figure 4.13: Graph of how the run-time varies for each run. Each run has different input, but all approaches uses the same input for the same run.

The simulation shows that the ROST algorithm is similar to or better than existing bin-packing approaches in run-time. A plus with the ROST algorithm is that it is better at distributing jobs across the available resources instead of packing one by one machine. This is an advantage that would have provided even better results if the simulation also added a penalty in run-time for all jobs on hosts with low leftover capacity. The results from the simulation is also a low scale example to demonstrate the different approaches. In reality, a job won't take a few seconds, but often several minutes and the queue time

and workflow run-time will therefore be a lot longer than in this simulation. Saving a second might not seem like a lot, but when this is used in a larger scale, the time saved will be more significant.

4.10.8 Further development of the simulation

In most exploratory assignments there are limited time, which often is part of drawing the line to where the research has to stop. While creating the simulation, a lot of further exploration potential and improvement potential was discovered and with more time, these would have been prioritized. The first step to improving the simulation is to include a run-time penalty to all jobs on a host with little left-over capacity. This will reward approaches which distribute jobs more evenly such as the worst-fit and the ROST algorithm.

Furthermore, the input jobs variation is limited. In other scenarios the job sizes might even be bigger than the smallest hosts can handle, and how the different algorithms will be affected by this is interesting to investigate further. Different job input has as shown caused variations to the time used by the individual algorithm, but it can also affect how well it performs compared to the others. Providing the simulation with hard-coded input flows with different type of edge cases is also interesting to investigate. Observing these results to see how the ROST algorithm's results are affected by such edge cases in comparison to the other approaches is very interesting.

Another addition to the simulation would be to increase the case scenario to be more similar to real scenarios with more and bigger jobs and running it continuously to observe the results over time. It would also be interesting adding a delay every now and then before new jobs enter the queue to observe how changing back to the worst-fit algorithm would affect the ROST algorithm results.

Investigating the potential around resource sharing would also be interesting. If one job use more resources in the first minute, while another job uses more resources in the

last minute, placing these together will create a synergy. Furthermore, as the bins can be printed continuously during the placement of the items, it can be interesting to create jobs and bins with different dimensions than the ones chosen and from there observe which of them are bottlenecks in the system.

One thing that is lacking in the simulation is a measuring unit for optimal placement. The run-time of a complete pipeline is a good unit for comparing how this will work in a CI system, but it would be interesting to get more data such as a unit to investigate optimal placement. This would provide a better way of analysing how each item is placed using the different approaches. This simulation is seen in a macro view, where the pipeline is analysed as a whole, but investigating each individual bin and job more closely could lead interesting observations.

4.11 Chapter summary

This chapter started with deciding what pieces of software testing would fit as an item and what dimensions would be used in a bin. A job was selected as the item and the dimensions were set to CPU and RAM as these were discovered to be the most common bottlenecks in a CI system. From there, different bin-packing approaches were analysed to find one that fits with automatic software testing. Many differences between the two concepts were discovered leading to the conclusion that not one algorithm would fit perfectly by itself. The approach chosen to further investigate was a combination of two bin-packing algorithms, the worst-fit and the best-fit depending on the queue state. A little inspiration were also used from the optimal job scheduling approach to top it all off. A case scenario were created and a simulation to test the approach. It was then compared to a best-fit, worst-fit, a first-fit combined with job scheduling approach and a hard-coded approach based on the solution that CI systems use today.

The ROST algorithm starts with an optimal job scheduling approach which puts one item in each bin, starting with the fastest one. When each bin has one item, the worst-fit algorithm is used as long as there is more room to fit more items. When the next-item

don’t fit, the algorithm is switched to a best-fit approach, to fully utilize the rest of the bin space for the upcoming items.

The ROST algorithm was shown through the simulation to be just as good or better than the other approaches when the run-time of a full workflow was measured. The simulation, however, lacks the run-time penalty on items which is running in bins with low capacity, which is assumed to provide better results for the worst-fit and ROST algorithm, because they spread items out more evenly among the hosts. The simulation could still be improved to cover more cases and show more detailed results, but it overall gave an impression of how the ROST algorithm could work in a continuous integration system.

5 Discussion

In this chapter, the path that has been selected will be discussed and reflected upon. This section is more about the journey taken on this exploratory adventure, than the actual results. How and why did we end up here?

5.1 A bin-packing approach, colored by software testing

The simulation shows that using bin-packing in general improves the results by a lot compared to a hard-coded approach, which is the common way of doing things today. The ROST algorithm shows even better results than the pure bin-packing approaches by some. The simulation is, however, a very minimized example, and the differences may be even bigger in a real size scenario. Saving a second in an approach that takes around 30 seconds can in a bigger scale be minutes and as Bell from CircleCI stated in their machine vs queue cost analysis: ”allowing queue times of more than 1 minute is like valuing your developers’ time at less than a dollar an hour” (Bell, 2016). The drastic reduction in run-time from the hard-coded approach to the bin-packing approaches shows that if more items fits in each bin, it will provide more space for more jobs, therefore reducing queue time and increase efficiency.

The results are overall colored by the fact that I choose to hold on to the bin-packing concept even though it at times seemed to not properly fit into this assignment. The solution in the end seems to fit, but during the process there were a lot of times that the path I had chosen was in doubt. At some point during the research period, it was discovered that a regular bin doesn’t match with how computer resources are packed. Thankfully the vector bin-packing concept was discovered shortly after, putting the assignment back together after it nearly fell apart. Later on, when the job scheduling problem was discovered, the bin-packing approach again looked to be less relevant than job scheduling due to the time concept of jobs finishing and being removed after completion. Instead of completely changing paths, however, it was decided that the time spent on exploring bin-packing couldn’t go to waste, and the goal was therefore to find a

model that could use both of the approaches.

Something else that strongly affected the results was the desire to stay true to the software testing concept over forcing bin-packing to fit straight in. The differences between bin-packing and software testing were many, causing more doubt around the whole bin-packing concept, but in the end, it was incredibly important to not make a fake scenario where everything would fit perfectly and instead tweak the algorithms to fit software testing over the other way around.

5.2 A challenging, but educational journey

The doubts to the concept experienced along the way also caused doubt around the exploring research approach. Nevertheless, in the end, an exploratory approach was exiting as I got challenges along the way to handle, and had to be creative to find new ways of approaching the initial idea. The results look very promising for further exploration and implementation, and the model and approach is more interesting than simply using an existing bin-packing algorithm within a CI-system. An exploratory assignment opens up more room to be creative, and discoveries can be more surprising than in a comparative study where there is a bigger understanding of what will happen.

Choosing an exploratory assignment has also put its marks on the results. There was as mentioned several times that the path taken was doubted and it was time consuming to continuously discover new concepts that might fit better or concepts by bin-packing and continuous integration that didn't go well with each other. The exploratory approach was interesting and a lot of new discoveries were made along the way, but the short time available forced the assignment back onto a straighter path than expected in the beginning. From experience with video games, it is common to start by selecting the wrong path to make sure that every part of the map is properly explored instead of rushing to the goal. It is often tempting to do the same while approaching a subject in an exploratory way. Unfortunately, the time period available were too short to be able to

investigate every corner. The exploration into the task had to come to an end due to lack of time.

It was challenging trying to find a whole new way of approaching the problem with such short time. One issue with an exploratory assignment is that you don't know what you will find, or if something usable is discovered at all. The exploration had to come to a halt after discovering one suitable approach as there was no time to look further after better approaches. There are several other bin-packing algorithms for instance that haven't been properly investigated and we also barley touched the surface of the job scheduling approach. There are also plenty of potential bin dimensions that could have been discussed more closely or even tested out in the simulation, but time is a constraint that is hard to get around.

Though challenging, I still think the exploratory approach was the right way of approaching this subject. In the beginning, it didn't seem right to see this as an exploratory assignment, as the immediate idea was that using bin-packing would be perfect and that it would solve all the problems. On the contrary, when looking at the details, it became clear that this wasn't a perfect match. The project started to look more like puzzle-solving trying to solve new problems using existing tools (Kuhn, 1962). The time frame was challenging, but the results became, in my opinion, more interesting than they might have been with a comparative approach. In the end, the path taken lead to an interesting assignment, and I would have chosen the same again if I started over, though having more time could have pushed the project even further towards implementation.

5.3 Combining the concepts: software testing and bin-packing

Bin-packing has been used for many years, mostly for physical packing, but also within the IT world. It has, however, not been used within software testing. The reason is, I believe, due to the mismatches between the two concepts and the complex choices that have to be made before it can easily be used. When experiencing many challenges to using a concept and implementing it into a solution, it might take a while before it is

prioritized by the CI providers. However, the simulation shows that even using pure bin-packing algorithms with adding the job removal concept and taking the job size variation into account improves the situation by a lot because the host machine resources are better utilized. Therefore, it might seem that prioritizing this will provide much value for the customers, however, the concept needs to be explored a bit further before it is easier to use for the customers.

Software testing using a continuous integration system seen through bin-packing goggles is about optimally placing jobs onto runners or agents in order to utilize all the resources as efficiently as possible. The differences, however, are many. Firstly, the jobs are being removed after they have finished their tasks compared to regular bin-packing where the goal is simply to pack all items and close the box. Secondly, the job sizes may vary over time, which is also highly unlikely within regular bin-packing. Third, there is a finite number of bins where the bin sizes often vary in sizes. Lastly, the resources cannot be stacked like in regular bin-packing, the problem therefore has to be approached as a vector bin-packing problem. All these differences makes it hard to approach this as a simple bin-packing problem and all of these aspects have to be taken into consideration to make the concepts fit. It is hard to stay true to the directions set purely by bin-packing using one algorithm to place the items. This assignment has therefore taken a turn, it's own path where the concept "bin-packing" still has been used, but with a twist to make sure to address the issues with combining it with software testing.

The complex choices along the way is also part of why I believe this combination hasn't been promoted much elsewhere. Although the goal is a utopia where every item fits perfectly alongside the others, the reality doesn't always reflect this. Choosing the best algorithm for this task was difficult as a single one didn't perfectly fit, and in the end a combination of two algorithms was chosen as the best option. Nevertheless, there exist other algorithms that haven't been addressed in this paper, which might be an even better fit. The trigger conditions to switch between the algorithms also has to be selected

based on conditions which might vary for individual companies. On top of choosing algorithms, the most complex choice lays with setting the individual resource requirements for each individual job. There could be numerous jobs in a pipeline and each job has to be evaluated for each individual resource to best utilize the resources. This brings us to another complex choice, the dimensions to use. There are plenty of possible dimensions, some may be bottlenecks, and some may not be and if too many is selected, the complexity of selecting the best place to put an item increases as each dimension has to be evaluated. Which ones are more important to utilize better and which ones can be neglected? This is a difficult task and the choices may vary from company to company and even from job to job based on what the jobs are doing.

Even though the resources will be better utilized in the end, it may seem that the path to achieve it is highly resource demanding and wont pay off for a very long time where the alternative is adding more bins, or in this case, hosts. This is, however, a costly investment both financially, but also environmentally and utilizing all resources better should be a prioritization. Nevertheless, the concept needs to grow into something more concrete before it is optimal to implement it as there are still loose threads and problems that aren't solved.

The results of this assignment is also based on the decision to focus on self-hosted on-premise hardware as hosts. The cloud is as mentioned more flexible and scalable and can already utilize the resources well by creating runners or agents based on the job's needs instead of having a constant size. Cloud instances, however, aren't free, and while it might be cheaper the first couple of years, owning the hardware is usually cheaper in the long run. As mentioned, there are also security aspects that needs to be considered while using the cloud as the developers don't fully have control of the cloud units, where the data is stored and where the data is transported. Another question is, how does the CI system handle cloud instances that have finished their jobs, do they stay online and under utilized or are they shut off before creating a new instance to fit perfectly for the next job? Automatically adjusting a cloud instance size as the job size

varies can also be difficult. Cloud auto-scaling might therefore also be interesting to explore further and optimize, but this hasn't been a focus in this project.

5.4 Towards improved software testing

In the end, is using bin-packing to better utilize resources within a continuous integration system feasible? The short answer is: ”no”. There are a lot of differences between the pure concept of bin-packing and software testing, but with some adjustments, the answer could be, ”yes”. The concept alone doesn't cover all the aspects required by software testing, but if one take the finite bins of different sizes, the item size variations, the job removal and the bin size variation into account, it seems possible. The implementation, however, can be highly complex due to all the choices that has to be made. Selecting dimensions, algorithms, trigger conditions and job requirements requires a lot of time and observations of the system to make the best choices.

In a world with limited resources, it is always important to focus on utilizing resources better. In this case, it comes down to making developers more effective and utilizing computer hardware better. Developer waiting time is an expensive resource and hardware itself is also a costly resource. Buying more hardware is an expensive and environmentally negative solution to decrease CI job queue time and better resource utilization should be a priority.

This project should be seen as a part of a bigger project where the end goal is implementing an easy to use, resource optimizing solution into a CI system. There are still more work that needs to be done, more decisions that needs to be made and more paths that needs to be explored.

5.5 Future work

There are often several other factors that come to play when selecting an agent for a job. For instance different software, OS, or other hardware than what was explored in this

assignment. Typical bottlenecks in a CI system is something that can be explored further, but how to handle it in the setup typically won't differ much from what has been discussed in this assignment as Kubernetes has the option to add non-Kubernetes built-in resources.

One of the biggest problems discussed in this assignment is the difficulties of assigning the correct amount of resources to each individual job. Something that would make this approach more usable and easier to implement, is to add a machine learning concept to the adjustable variables. The system can learn and adjust by itself based on wishes set by the developers. Each job can for instance have some standard resource limits and based on the outcome, the system can either give more or less resources to the jobs. If the developers want the jobs to run as fast as possible, the system can adjust until it finds the point where adding more resources doesn't change anything and set the limit there. On the other hand, if the developers want the jobs to just have enough resources to run as much as possible at the same time, the resource limit can be set just beyond the breaking point where less resources would cause errors. Adding this to the system will make the approach much more usable for the developers as it is a lot easier to set up. This will remove much of the complexity, and the usefulness of optimizing the resource utilization is therefore much easier to vouch for.

The job size variation has only been briefly touched upon in this assignment. The job size variation is an important focus when it comes to utilizing all resources in the best possible way. To properly decide how much space a job require, it is important to see how the resource usage varies over time in a continuous integration system. If, for instance, resource usage goes up at some point, and other jobs have filled the remaining space on the host already, the job requiring more resources might at worst get cancelled. It is therefore important to figure out when to allocate more or less resources to a job to make sure it has enough available at the point when the size changes while still not setting aside resources which can be used by other jobs. How to handle this is something that requires more research as this approach doesn't consider increasing

resource requirements.

In this paper I've explored the usefulness of using bin-packing in system-testing, but actually implementing it is a job for the future. The possibility of implementation has been briefly discussed, it seems possible, but issues often don't present themselves before the implementation process. Running the algorithms in real scenarios might also unveil advantages or disadvantages that haven't been discovered through this process, and watching the project come to life and observing it over time will be interesting.

There are other ways of making a continuous integration system more effective than utilizing resources better. Adding more hosts to the system has been mentioned, but the downsides to this is the cost both financially and environmentally. Another simpler way of increasing efficiency is to make sure the test order is optimal. It can for instance be better to place tests that fail often early in the pipeline to stop all other tests before they use resources unnecessarily. Placing jobs that have a lot of dependencies early can also grant advantages such as queuing more jobs faster and therefore possibly complete the pipeline faster. Splitting a job into smaller jobs is also something that can improve the situation, as it's easier to fit more sand in a jar than rocks or pebbles. This, on the other hand, requires that the hosts don't have hard-coded limits to improve the situation. Reducing the amount of bugs or test the code before it is run through the CI system will also ensure that the pipeline fails less often and therefore cause fewer necessary runs of a pipeline. Another improvement which will increase CI efficiency is to make sure that unnecessary plugins and software isn't installed every time a job runs as the setup usually is what takes the most time. Despite all the other improvement areas mentioned, using bin-packing for placing jobs optimally onto hosts seems to be a huge improvement in comparison to a hard-coded approach and can therefore be a huge step in the right direction of increasing efficiency.

Using the cloud has also been briefly discussed in this assignment, and as the cloud is more flexible than using on-premise resources, it might be interesting to investigate what

advantages using the cloud as agents instead of on-premise hosts will grant. There might, however, be resource utilization issues using the cloud as well, and on top of that, a lot of developers prefer or require on-premise hardware to run their pipeline for different reasons. Some companies create their own hardware and have to test specifically on that, or the price and security factor can be the reason why some choose to use the CI system with on-premise hardware. Nevertheless, it might be interesting to see if there are improvement potential to using bin-packing for better resource utilization in a cloud setting as well.

6 Conclusion

The goal of this project was to: *explore the feasibility of applying bin-packing for optimizing resource utilization in software testing.* Its possible advantages and disadvantages has been critically investigated and a novel model, which addresses relevant shortcomings has been present.

A non-trivial answer to selecting a placement algorithm in a CI system was hard to determine. A lot of bin-packing algorithms exist and some of them have been discussed in this project. The many differences between bin-packing and software testing did at many times cause doubt during the process, but solutions were found which pushed the project forward. These differences lead to the realization that a single bin-packing algorithm isn't a perfect fit into the software testing concept, but that combining several approaches can address the gaps identified in this paper.

One of the most significant differences between bin-packing and software testing is that in software testing, items are removed from a bin after a certain amount of time. When a software test finishes, it opens up room to place new items in the bins. In addition, software test job requirements may vary over time, whereas in bin-packing, items are rigid and constant. Furthermore, bin-packing is about packing the given items into as few bins as possible, but in software testing, it is about finishing the most jobs as fast as possible. Finally, since software testing in CI is about optimizing several resources rather than just physical space, a vector-based bin-packing method must be applied.

This projects, like any research, has opened many new doors, but also closed others. It has been shown that bin-packing in general will shorten the required time to run a continuous integration pipeline and that the model presented, the ROST algorithm, is just as good or better than existing bin-packing algorithms.

The ROST algorithm is a combination of several approaches. The first step of the approach is to use an optimal job scheduling approach to put one item in each bin, starting with the fastest one. Secondly, the bin-packing algorithm called worst-fit is used to distribute items evenly out on the hosts. When the next item doesn't fit in any bin, the algorithm is switched to another bin-packing algorithm, the best-fit algorithm. This is used to place the final jobs onto hosts in order to optimally utilize the remaining resources. If, however, the next job fits in more than half of the available bins, the approach switches back to the worst-fit algorithm.

There are still some challenges to handle before implementing this into a CI system. Every job has to be assigned an amount of resources, and setting the correct limits are critical to fully utilize all available resources. This requires a lot of knowledge about the jobs and where the critical resource limits are. Further development is required to investigate if this could be automated using machine learning.

In the end, the approach according to this research, seem implementable as the auto-scaling using Kubernetes already provides most of the building blocks. This approach has also shown, through the simulation, to provide great advantages to better resource utilization and workflow efficiency in comparison to the solution most commonly used for CI systems today.

References

Amazon. (2022). *Best sellers in computer memory.* Retrieved from https://www.amazon.com/Best-Sellers-Computer-Memory/zgbs/pc/172500

ArsTech. (2020). *10 best selling cpu amazon as of february 2020.* Retrieved from https://arstech.net/best-selling-cpu-amazon/

Basu, S. (2017). Continuous integration: Its history and benefits. *DevOps Zone*. Retrieved from https://dzone.com/articles/continuous-integration-and-its-whereabouts

Bell, K. (2016). *Simulating auto scaling build clusters part 1: The mathematical justification for not letting your builds queue.* Retrieved from https://circleci.com/blog/mathematical-justification-for-not-letting-builds-queue/

Bhardwaj, M. (2021). *Addressing slow performance in jenkins.* Retrieved from https://earthly.dev/blog/slow-performance-in-jenkins/

Boyar, J., Epstein, L., Favrholdt, L. M., Kohrt, J. S., Larsen, K. S., Pedersen, M. M., & Wøhlk, S. (2006). The maximum resource bin-packing problem. *Theoretical Computer Science, Volume 362*. Retrieved from http://math.haifa.ac.il/lea/lbp.pdf

Buggy, P. (2020). *What to do when you've got too much to do (rocks, pebbles, and sand).* Retrieved from https://mindfulambition.net/big-rocks-first/

Bulao, J. (2022). *How many companies use cloud computing in 2022? all you need to know.* Retrieved from https://techjury.net/blog/how-many-companies-use-cloud-computing/#gref

Celiku, L. (2021). Towards continuity-as-code. *Thesis submitted for the degree of Master in Network and System Administration*. Retrieved from https://www.duo.uio.no/bitstream/handle/10852/87276/5/Lea_Celiku_NSA_Master_Thesis.pdf

CloudBees. (2022). *Remoting best practices for cloudbees ci on traditional platforms.* Retrieved from https://support.cloudbees.com/hc/en-us/articles/

115002654991-Remoting-Best-Practices

de Niz, D., & RajKumar, R. (2006). Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems 2*. Retrieved from https://www.researchgate.net/publication/220309658_Partitioning_bin-packing_algorithms_for_distributed_real-time_systems

Fisher, C. (2018). Cloud versus on-premise computing. *American Journal of Industrial and Business Management*. Retrieved from https://www.scirp.org/html/7-2121263_87661.htm

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Retrieved from http://cope.eecs.oregonstate.edu/papers/OpenSourceCIUsage.pdf

Horowitz, D. (2020). *Cpu cores: How many do i need?* Retrieved from https://www.hp.com/us-en/shop/tech-takes/cpu-cores-how-many-do-i-need

Hwang, H.-C., Park, J., & Shon, J. G. (2016). *Design and implementation of a reliable message transmission system based on mqtt protocol in iot.* Retrieved from https://www.researchgate.net/figure/CPU-usage-graph-per-each-process_fig2_303888951

Jenkins. (2022). *Executor starvation.* Retrieved from https://www.jenkins.io/doc/book/using/executor-starvation/

Jin, H., Pan, D., Xu, J., & Pissinou, N. (2012). Efficient vm placement with multiple deterministic and stochastic resources in data centers. *2012 IEEE Global Communications Conference (GLOBECOM), 2012*. Retrieved from https://users.cs.fiu.edu/~pand/publications/12globecom-hao.pdf

Karthik, C., Sharma, M., Maurya, K., & Chandrasekaran, K. (2016). Green intelligence for cloud data centers. *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*. Retrieved from https://ieeexplore.ieee.org/document/7507965

Kingston. (2019). *How much memory do you need to run windows, mac os x, or linux*

apps? Retrieved from
https://www.kingston.com/en/memory/desktop-laptop/memory-assessor

Kubernetes. (2021). *Pods.* Retrieved from
https://kubernetes.io/docs/concepts/workloads/pods/

Kubernetes. (2022). *Resource management for pods and containers.* Retrieved from
https://kubernetes.io/docs/concepts/configuration/
manage-resources-containers/

Kuhn, T. S. (1962). *The structure of scientific revolutions*.

Kumar, P. R., Raj, P. H., & Jelciana, P. (2018). Exploring data security issues and solutions in cloud computing. *Procedia Computer Science Volume 125*. Retrieved from https://www.sciencedirect.com/science/article/pii/S1877050917328570

Kumaraswamy, S., & Nair, M. K. (2019). Bin packing algorithms for virtual machine placement in cloud computing: A review. *International Journal of Electrical and Computer Engineering 9*. Retrieved from
https://www.researchgate.net/publication/332546043_Bin_packing
_algorithms_for_virtual_machine_placement_in_cloud_computing_A_review

Madel, K. (2018). *Autoscaling jenkins agents with kubernetes.* Retrieved from
https://kurtmadel.com/posts/cicd-with-kubernetes/
autoscaling-jenkins-agents-with-kubernetes/

Meyer, M. (2014). Continuous integration and its tools. *IEEE Software, Volume: 31*. Retrieved from https://ieeexplore.ieee.org/document/6802994

Mohammad, S. M. (2020). Continuous integration and automation. *International Journal of Creative Research Thoughts Volume.4*. Retrieved from
https://deliverypdf.ssrn.com/delivery.php?ID=
535088111116103122004000090123084122000020077035034062072085114067098082127118102090
pdf&INDEX=TRUE

Rawitz, D., & Patt-Shamir, B. (2012). Vector bin packing with multiple-choice. *Discrete Applied Mathematics, Volume 160*. Retrieved from
https://reader.elsevier.com/reader/sd/pii/S0166218X12000819?token=

CDE670E2DF8524C49462A648AC45B7D13EE16C5B6625884F2E2FFFCE8AF13907E26BD1F1EEF421B6E3C
eu-west-1&originCreation=20220220165504

Santamaria, M. (2018). Developers are using ci more than cd, report finds. *SDTimes*. Retrieved from https://sdtimes.com/cicd/developers-using-ci-cd-report-finds/

Schultz, J. (2017). *Five causes of performance bottlenecks in it.* Retrieved from https://www.helpsystems.com/blog/five-causes-performance-bottlenecks-it

Sirois, S. (2018). *What is processor speed and why does it matter?* Retrieved from https://www.hp.com/us-en/shop/tech-takes/what-is-processor-speed

srisuk, T. (n.d.). *Blue puzzle is not complete.* Retrieved from https://www.shutterstock.com/image-vector/blue-puzzle-not-complete-1095560672

Szoke, A. (2010). Bin-packing-based planning of agile releases. *Communications in Computer and Information Science 69*. Retrieved from https://www.researchgate.net/publication/225265111_Bin-Packing-Based_Planning_of_Agile_Releases

Wikipedia. (2022). *Bin packing problem.* Retrieved from https://en.wikipedia.org/wiki/Bin_packing_problem

Yalcin, M. I. (n.d.). *Colorful tetris game vector design.* Retrieved from https://www.shutterstock.com/image-vector/colorful-tetris-game-vector-design-1632748039

7 Appendix

7.1 The simulation

```
1  from dataclasses import dataclass
2  import threading
3  import time
4  from timeit import default_timer as timer
5  from datetime import timedelta
6  import random
7  from prettytable import PrettyTable
8  import os
9
10
11 class _Bin():
12     def __init__(self, cpu, ram, num, clock_speed):
13         self.full_cpu_capacity = cpu # The full capacity is set
               as well as one variable that will vary based on item
               content
14         self.full_ram_capacity = ram
15         self.cpu = cpu
16         self.ram = ram
17         self.num = num
18         self.items = []
19         self.clock_speed = clock_speed
20
21
22     def __repr__(self):
23         return str(self)
24
```

```
25
26      def __str__(self): # Draw bin showing capacity
27          bin_drawing = ("\n" + " " + "_" * self.full_cpu_capacity
              + "\n")
28          for i in range(self.full_ram_capacity):
29              if i == self.ram:
30                  bin_drawing += "|" + "¯" * (self.
                      full_cpu_capacity - self.cpu) + "|" + " " * (
                      self.cpu-1) + "|" + "\n"
31              elif i > self.ram:
32                  bin_drawing += "|" + " " * (self.
                      full_cpu_capacity - self.cpu)+ "|" + " " * (
                      self.cpu-1) + "|" + "\n"
33              else:
34                  bin_drawing += "|" + " " * (self.
                      full_cpu_capacity) + "|" + "\n"
35          bin_drawing += (" " + "¯" * self.full_cpu_capacity + "\n"
              )
36          return bin_drawing
37
38
39      def remove_item(self, item):
40          runtime = item.runtime/self.clock_speed
41          time.sleep(runtime/2) # after half the time reduce
              resource usage
42          cpu_reduced = False
43          ram_reduced = False
44          if item.cpu > 1:
45              cpu_reduced = True
46              self.cpu += 1
```

```
47          if item.ram > 1:
48              ram_reduced = True
49              self.ram += 1
50          time.sleep(runtime/2)
51          self.items.remove(item)
52          if cpu_reduced:
53              self.cpu += (item.cpu-1)
54          else:
55              self.cpu += item.cpu
56          if ram_reduced:
57              self.ram += (item.ram-1)
58          else:
59              self.ram += item.ram
60
61
62      def place_item(self, item):
63          self.items.append(item)
64          self.cpu -= item.cpu
65          self.ram -= item.ram
66          countdown = threading.Thread(target=self.remove_item,
              args=(item,))
67          countdown.start()
68
69
70  class _Job():
71      def __init__(self, cpu, ram, num, job_length_factor):
72          self.cpu = cpu
73          self.ram = ram
74          self.num = num
75          self.runtime = (cpu+ram)*job_length_factor
```

```
76
77      def __str__(self):
78          return f"Item {self.num} has requirements: cpu:{self.cpu
                }, ram:{self.ram}"
79
80
81  @dataclass
82  class _BinVariables:
83      available_bins: list
84      bin_items = []
85
86
87  def _create_tests(job_length_factor = 10) -> None: # Random
        generate 20 jobs within the given interval values
88      for i in range(20):
89          _BinVariables.bin_items.append(
90              _Job(cpu = random.randint(1,3), ram = random.randint
                    (1,4), num=i, job_length_factor=job_length_factor)
91          )
92
93
94  def _create_bins() -> None:
95      _BinVariables.available_bins = [
96          _Bin(cpu = 6, ram = 8, num = 1, clock_speed=4.2),
97          _Bin(cpu = 8, ram = 16, num = 2, clock_speed=4.4),
98          _Bin(cpu = 8, ram = 16, num = 3, clock_speed=4.1),
99          _Bin(cpu = 12, ram = 32, num = 4, clock_speed=4.6)
100     ]
101
102
```

```
103 def _best_fit() -> None:
104     all_bins = _BinVariables.available_bins
105     for item in _BinVariables.bin_items:
106         min_remaining_space = 999999
107         fitting_bin = None
108         while fitting_bin == None:
109             for agent in all_bins:
110                 if item.cpu < agent.cpu and item.ram < agent.ram:
111                     remining_space = (agent.cpu - item.cpu) + (
                            agent.ram - item.ram)
112                     if remining_space < min_remaining_space:
113                         min_remaining_space = remining_space
114                         fitting_bin = agent
115             if fitting_bin != None: # found best bin
116                 fitting_bin.place_item(item)
117                 # _create_pretty_table()
118             else: # Item doesn't fit in any bin
119                 time.sleep(1)
120 
121 
122 def _worst_fit() -> None:
123     all_bins = _BinVariables.available_bins
124     for item in _BinVariables.bin_items:
125         max_remaining_space = -1
126         worst_fit_bin = None
127         while worst_fit_bin == None:
128             for agent in all_bins:
129                 if item.cpu < agent.cpu and item.ram < agent.ram:
130                     remining_space = (agent.cpu - item.cpu) + (
                            agent.ram - item.ram)
```

```
131                     if remining_space > max_remaining_space:
132                         max_remaining_space = remining_space
133                         worst_fit_bin = agent
134             if worst_fit_bin != None: # available bin
135                 worst_fit_bin.place_item(item)
136                 # _create_pretty_table()
137             else: # no available bin
138                 time.sleep(1)
139
140
141
142 def _optimal_job_scheduling() -> None: # A first-fit approach
    with bins sorted by clock speed
143     all_bins_sorted = sorted(_BinVariables.available_bins, key=
        lambda table: table.clock_speed, reverse=True)
144     for item in _BinVariables.bin_items:
145         placed_item = False
146         while not placed_item:
147             for agent in all_bins_sorted:
148                 if item.cpu < agent.cpu and item.ram < agent.ram:
149                     agent.place_item(item)
150                     placed_item = True
151                     # _create_pretty_table()
152                     break
153             if not placed_item:
154                 time.sleep(1)
155
156
157 def _two_jobs_in_each_bin() -> None:
158     all_bins = _BinVariables.available_bins
```

```
159     for item in _BinVariables.bin_items:
160         available_bin = None
161         while available_bin == None:
162             for agent in all_bins:
163                 if len(agent.items) < 2:
164                     agent.place_item(item)
165                     # _create_pretty_table()
166                     available_bin = agent
167             if available_bin == None:
168                 time.sleep(1)
169 
170 
171 def _rost_algorithm() -> None:
172     all_bins = _BinVariables.available_bins # Start with a job-
          scheduling (1 item in each bin, start with fastest bin)
173     all_bins_sorted = sorted(all_bins, key=lambda table: table.
          clock_speed, reverse=True)
174     all_items = _BinVariables.bin_items
175     for index in range(len(all_bins)-1):
176         all_bins_sorted[index].place_item(all_items[index])
177     worst_fit_algorithm = True
178     for index in range(len(all_bins)-1, len(all_items)): # Place
          the next items using worst-fit or best-fit depending on
          the remaining bin space.
179         if worst_fit_algorithm and not
              _switch_to_best_fit_algorithm(all_items[index]):
180             _place_worst_fit(item = all_items[index], all_bins =
                  all_bins_sorted)
181         elif worst_fit_algorithm and
              _switch_to_best_fit_algorithm(all_items[index]):
```

```
182             print("Switching to best fit")
183             worst_fit_algorithm = False
184             _place_best_fit(item = all_items[index], all_bins =
                    all_bins_sorted)
185         elif not worst_fit_algorithm and not
                _switch_to_worst_fit_algorithm(all_items[index]):
186             _place_best_fit(item = all_items[index], all_bins =
                    all_bins_sorted)
187         elif not worst_fit_algorithm and
                _switch_to_worst_fit_algorithm(all_items[index]):
188             worst_fit_algorithm = True
189             print("Switching to worst fit")
190             _place_worst_fit(item = all_items[index], all_bins =
                    all_bins_sorted)
191         else:
192             raise RuntimeError("Accidental case scenario occured,
                    one or more cases aren't covered in the if
                    statement")
193
194 def _switch_to_best_fit_algorithm(item: _Job) -> bool:
195     item_fits = False
196     for agent in _BinVariables.available_bins:
197         if item.cpu < agent.cpu and item.ram < agent.ram:
198             return False
199     return True
200
201
202 def _switch_to_worst_fit_algorithm(item: _Job) -> bool:
203     items_fits_in_num_bins = 0
204     for agent in _BinVariables.available_bins:
```

```
205         if item.cpu < agent.cpu and item.ram < agent.ram:
206             items_fits_in_num_bins += 1
207     if items_fits_in_num_bins > (len(_BinVariables.available_bins
        )/2): # If item fits in more than half of the bins
208         return True
209     return False
210
211
212 def _place_worst_fit(item: _Job, all_bins: list) -> None:
213     max_remaining_space = -1
214     worst_fit_bin = None
215     for agent in all_bins:
216         if item.cpu < agent.cpu and item.ram < agent.ram:
217             remining_space = (agent.cpu - item.cpu) + (agent.ram
                - item.ram)
218             if remining_space > max_remaining_space:
219                 max_remaining_space = remining_space
220                 worst_fit_bin = agent
221     if worst_fit_bin != None: # available bin
222         worst_fit_bin.place_item(item)
223         # _create_pretty_table()
224     else: # no available bin
225         raise RuntimeError("No bins available should not occur
            inside the worst-fit algorithm, fix errors")
226
227
228 def _place_best_fit(item: _Job, all_bins: list) -> None:
229     min_remaining_space = 999999
230     fitting_bin = None
231     while fitting_bin == None:
```

```
232         for agent in all_bins:
233             if item.cpu < agent.cpu and item.ram < agent.ram:
234                 remining_space = (agent.cpu - item.cpu) + (agent.
                        ram - item.ram)
235                 if remining_space < min_remaining_space:
236                     min_remaining_space = remining_space
237                     fitting_bin = agent
238         if fitting_bin != None: # found best bin
239             fitting_bin.place_item(item)
240             # _create_pretty_table()
241         else: # Item doesn't fit in any bin
242             time.sleep(1)
243 
244 
245 def _time_approach(function) -> None:
246     start = timer()
247     function()
248     empty_bins = 0
249     while empty_bins != 4:
250         empty_bins = 0
251         for agent in _BinVariables.available_bins:
252             if len(agent.items) == 0:
253                 empty_bins += 1
254     end = timer()
255     return timedelta(seconds=end-start)
256 
257 
258 def _create_pretty_table() -> None:
259     os.system('clear') # cls on Windows, clear on unix
260     table = PrettyTable()
```

```
261     table.field_names = ["Agent 1", "Agent 2", "Agent 3", "Agent
            4"]
262     agents = _BinVariables.available_bins
263     table.add_row([str(agents[0]), str(agents[1]), str(agents[2])
            , str(agents[3])])
264     print(table)
265
266
267 def _main() -> None:
268     _create_tests(job_length_factor=15)
269
270     _create_bins()
271     print("Running ROST algorithm approach")
272     time_rost_algorithm = _time_approach(_rost_algorithm)
273     print("ROST algorithm approach complete")
274
275     _create_bins()
276     print("Running hard-coded approach")
277     time_hard_coded = _time_approach(_two_jobs_in_each_bin)
278     print("Hard-coded approach complete")
279
280     _create_bins()
281     print("Running Worst-Fit approach")
282     time_worst_fit = _time_approach(_worst_fit)
283     print("Worst-Fit approach complete")
284
285     _create_bins()
286     print("Running Best-Fit approach")
287     time_best_fit = _time_approach(_best_fit)
288     print("Best-Fit approach complete")
```

```
289
290     _create_bins()
291     print("Running Optimal Job scheduling approach")
292     time_job_scheduling = _time_approach(_optimal_job_scheduling)
293     print("Optimal Job scheduling approach complete")
294
295     table = PrettyTable()
296     table.field_names = ["Hard-coded", "Worst-Fit", "Best-Fit", "
            Optimal Job-Scheduling", "ROST Algorithm"]
297     table.add_row([time_hard_coded, time_worst_fit, time_best_fit
            , time_job_scheduling, time_rost_algorithm])
298     print(table)
299
300
301 if __name__ == "__main__":
302     _main()
```