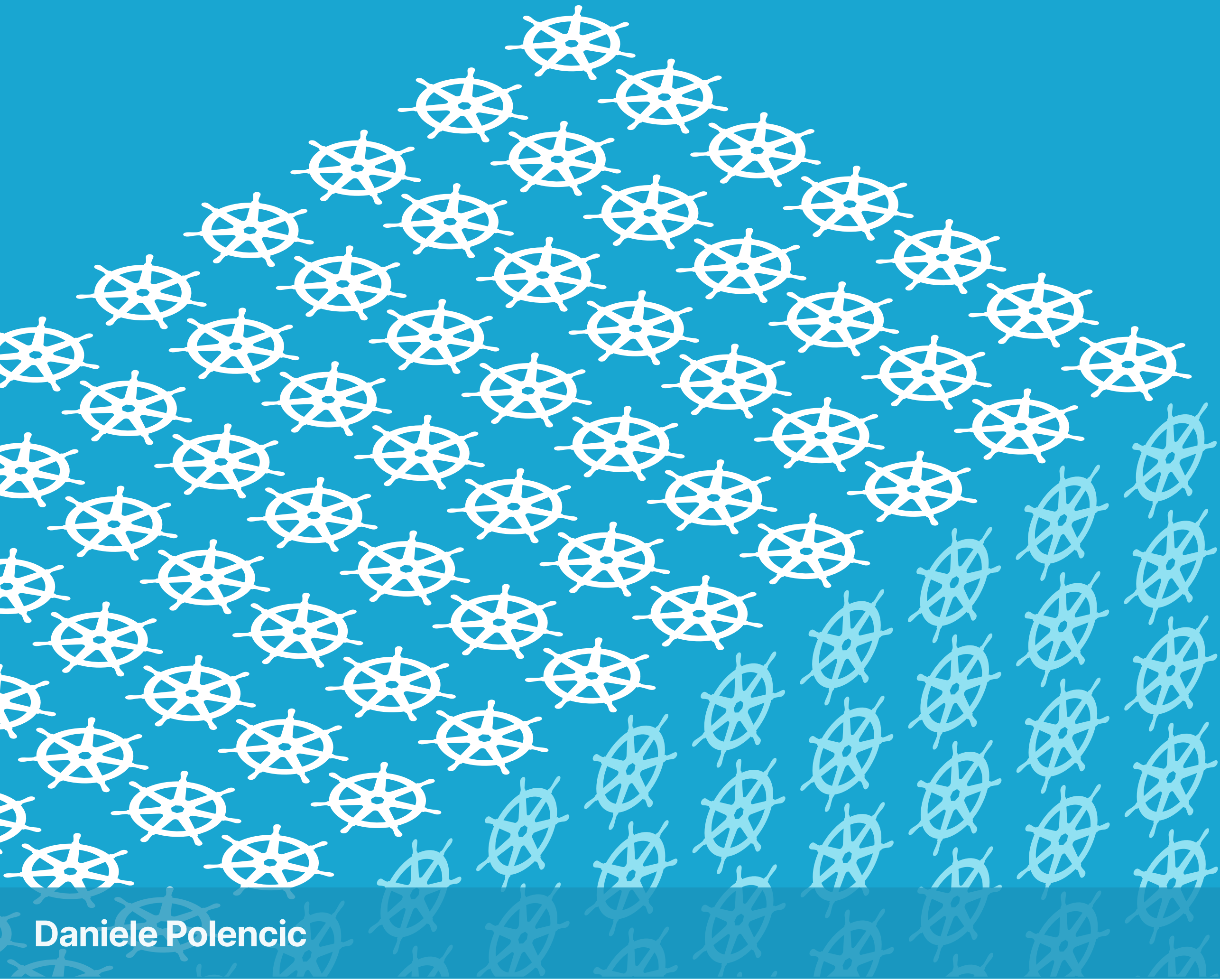
KUBERNETES FIRST STEPS

Learn how to package, deploy and scale applications with containers and Kubernetes.



Daniele Polencic

learnk8s

Table of contents

Chapter 1

Preface

Welcome to Learnk8s' Kubernetes first steps course!

This book is designed to take you on a journey and teach you why containers and Kubernetes are dominating modern development and deployment of applications at scale.

The book doesn't make any assumptions and covers the basics as well as advanced topics.

You will learn:

- What Kubernetes and containers are meant to replace (Chapter 1).
- How containers solve the problem of packaging, distributing and running apps reliably (Chapter 2).
- What is a container orchestrator and why you might need one (Chapter 3).
- The basic concepts necessary to deploy your apps on Kubernetes (Chapter 4).
- How to use a Kubernetes cluster (Chapter 5).
- What happens when an app is deleted or crashes in the cluster (Chapter 6).
- The full end-to-end journey of creating apps, packaging them as containers and deploying them on Kubernetes (Chapter 7).
- How different Kubernetes objects relate to each other and how you can debug them (Chapter 8).

That's the high-level plan.

Each chapter dives into more details.

The book is designed to be hands-on, so you can read it to the end, and re-read it while practising the command with a real cluster.

There are also hands-on challenges in Chapter 8 (with solutions).

When you complete all the challenges, you will be awarded a certificate of completion!

The book has three special chapters at the end:

1. A section designed to help you install the prerequisites.
2. A mastering YAML course — everything you need to master the configuration language used by Kubernetes.
3. Tips and tricks to become more efficient with kubectl — the Kubernetes command-line tool.

Prerequisite knowledge

This book tries to make as few assumptions as possible.

However, it's hard to cover everything singly and concisely.

To make the most out of this book, you should be familiar with:

- **A shell environment like Bash or Powershell.** You won't use complex commands, so a basic working knowledge is

- enough. If you haven't used Bash before and you want to start now, I recommend checking out Learn Bash the Hard Way by Ian Miell.
- **Virtual machines and virtualisation**. If you've used tools such as VirtualBox or Parallels to run other operating systems such as Windows, then you already know everything there is to know.
- **Web servers** such as Nginx, HAProxy, Apache, IIS, etc. If you've used any of these to host websites, that is enough to understand Kubernetes. If this is the first time you have heard about them, you can check out this introduction about Nginx.

In this journey, you are not alone

If at any time you are stuck or find a concept difficult and confusing, you can get in touch with me or any of the instructors at Learnk8s.

You can find us on the official Learnk8s Slack channel.

You can request an invite at this link.

Chapter 2

Infrastructure, the past, present and future

In the past few years, the industry has experienced a shift towards developing smaller and more focused applications.

It comes as no surprise that more and more companies are breaking down their static apps into a set of decoupled and independent components.

And rightly so.

Apps that are smaller in scope are:

1. **Quicker to deploy** — because you create and release them in smaller chunks.
2. **Easier to iterate on** — since adding features happens independently.
3. **Resilient** — the overall service can still function despite one of the apps not being available.

Smaller services are excellent from a product and development perspective.

But how does that cultural shift impact the infrastructure?

Managing infrastructure at scale

Developing services out of smaller components introduces a different challenge.

Imagine being tasked with migrating a single app into a collection of services.

When, for every application, you can refactor the same app in a collection of four components, **you have three more apps to develop, package and release.**

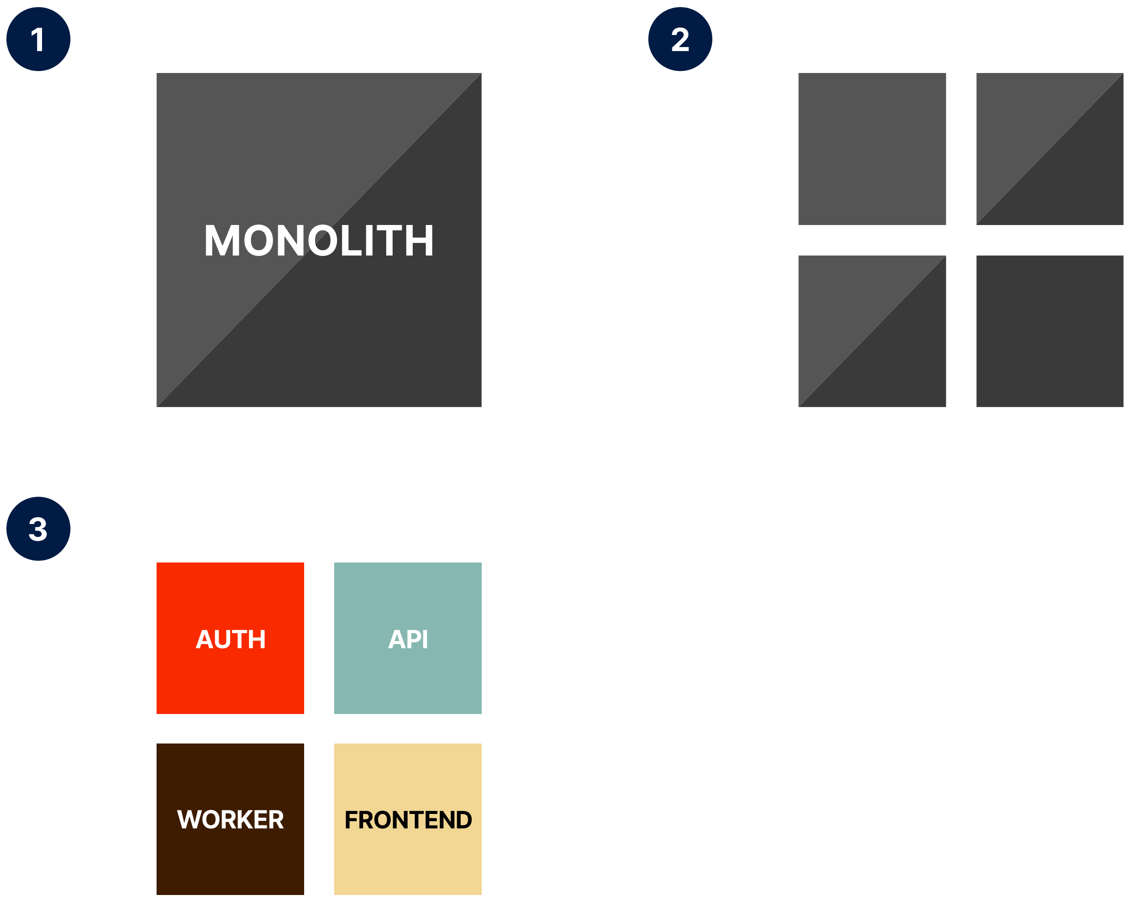


Fig. 1 Applications packaged as a single unit are usually referred to as monoliths.

Fig. 2 You might have developed applications as single units and then decided to break them down into smaller components.

Fig. 3 During the transition, you could have created three more apps that should be developed and deployed independently.

And it doesn't end there.

If you test and integrate your apps into separate environments, you might need to provision more copies of your environments.

For example, with only three environments such as development, staging and production, you might need to provision 12 environments — 9 more than you had to provision with a single app.

And that's still a conservative number.

It's common, even for smaller companies, to have dozens of components such as a front-end, a backend API, an authorisation server, an admin UI, etc.

Virtual machines

Your applications are usually deployed on a server.

However, it's not always practical to provision a new machine for every deployment.

Instead, it's more cost-effective to buy or rent a large computer and partition it into discrete units.

Each unit could run one of your apps isolated from the others.

Virtual machines are the primary example of such a mechanism.

With virtual machines, you can create virtual servers within the same server and keep your workloads isolated.

Each virtual machine behaves like a **real** server, and it has an operating system just like the real one.

When you develop apps that are smaller in scope, you might see a proliferation of virtual machines in your infrastructure.

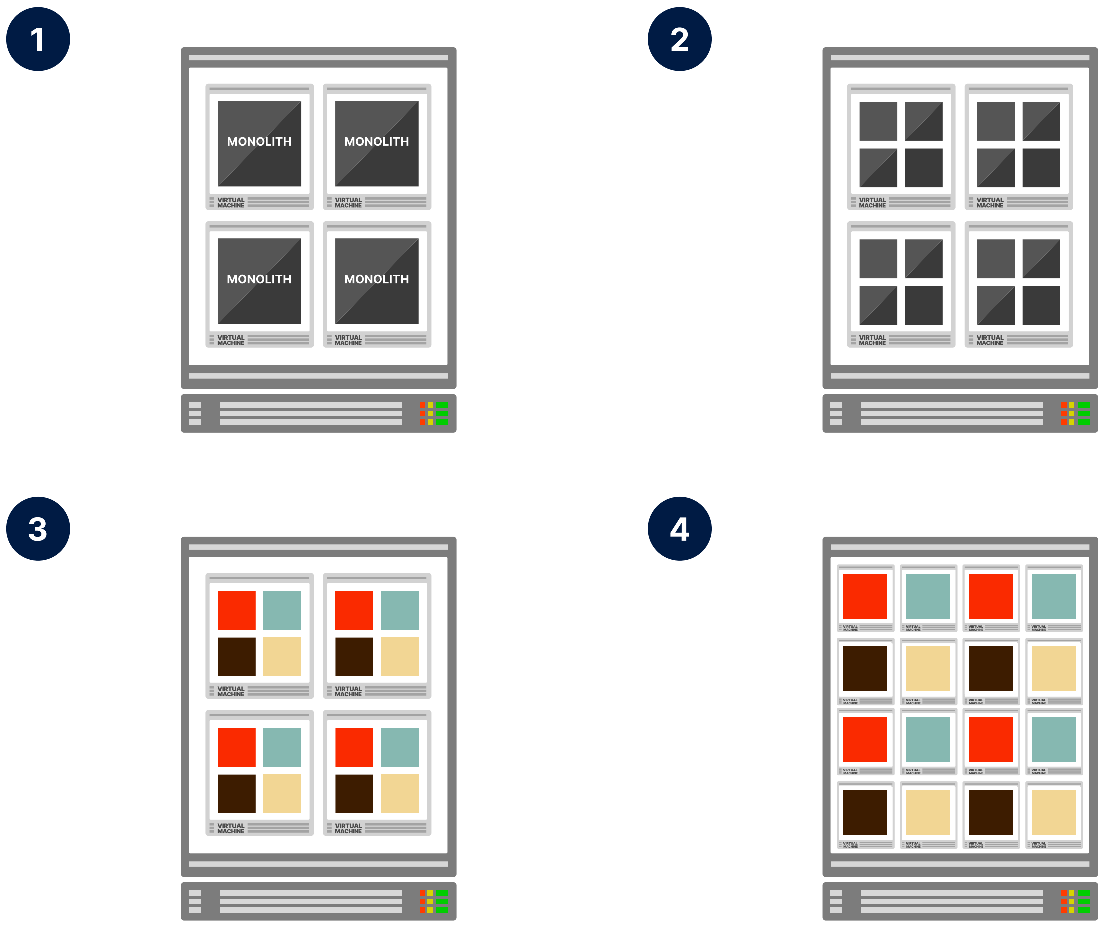


Fig. 1 Each server could run several virtual machines.

Fig. 2 As soon as you break the application into smaller components, you might wonder if it is wise to keep them in the same virtual machine.

Fig. 3 Perhaps it's best to isolate the apps and define clear boundaries between them.

Fig. 4 You might want to package and run those apps into separate virtual machines.

Virtual machines are widely used because they offer strong guarantees of isolation.

If you deploy an app in a virtual machine that shares a server with several others, the only thing you can see is the current virtual machine.

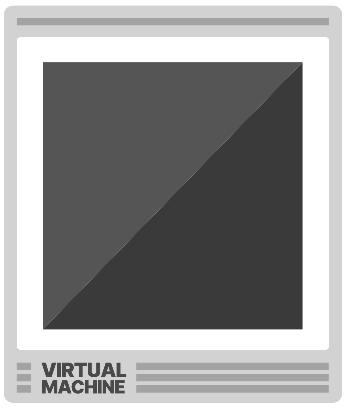
You can't tell that you are sharing the same CPU and memory with other apps.

But virtual machines have trade-offs.

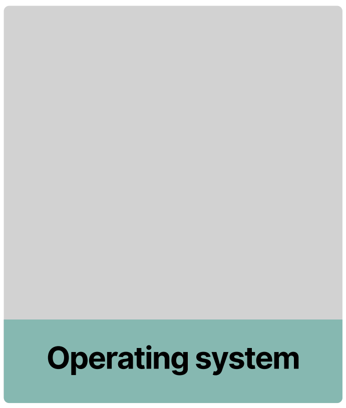
Resources utilisation and efficiency

Each virtual machine comes with an operating system that consumes part of the memory and CPU resources allocated to it.

1



2



3

Fig. 1 Consider the following virtual machine.

Fig. 2 Parts of the compute resources are used for the operating system.

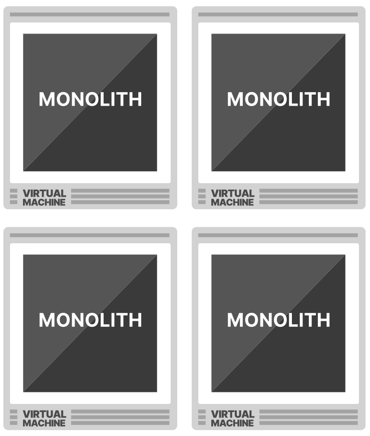
Fig. 3 The application can use the rest of the CPU and memory.

When you create a t2.micro virtual machine with 1GB of memory and 1 vCPU in Amazon Web Services, you are left with 70% of the resources after you account for the CPU and memory used by the operating system.

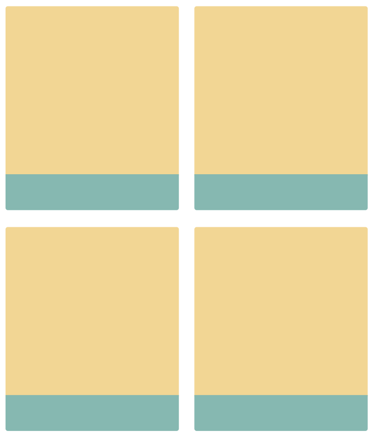
Similarly, when you request for one larger instance such as a t2.xlarge with 4 vCPU and 16 GiB of memory, you will end up wasting about 2% of the resources for the operating system.

For large virtual machines, the overhead adds up to about 2 to 3% — not a lot.

1



2



3

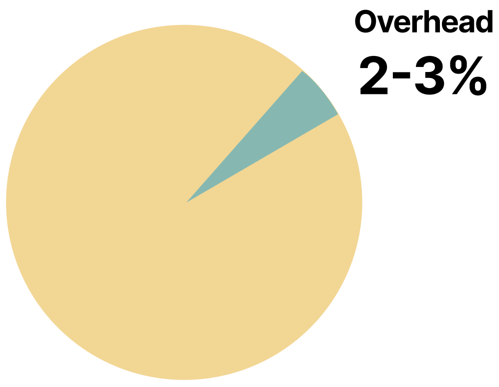


Fig. 1 When you develop fewer and larger apps, you have a limited number of virtual machines deployed in your infrastructure. They also tend to use compute resources with more memory and CPU.

Fig. 2 The percentage of CPU and memory used by the operating system in a large virtual machine is minimal — between 2 and 3%.

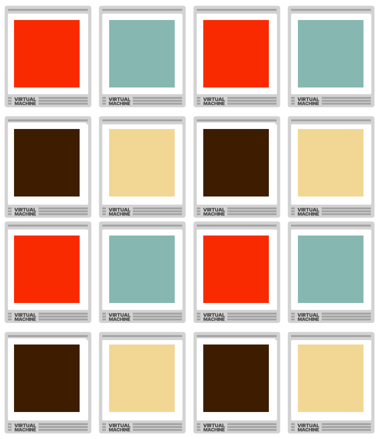
Fig. 3 However, that's not true for smaller virtual machines.

If you have four applications and decide to refactor them in a collection of 4 smaller components each, you have 16 virtual machines deployed in your infrastructure.

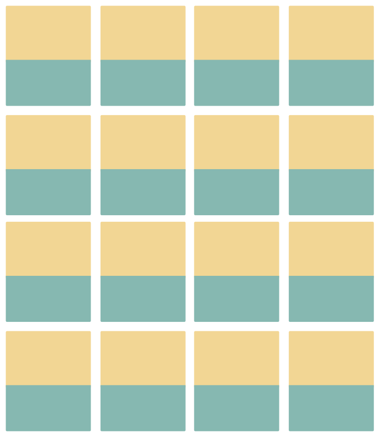
When each virtual machine has an operating system that uses 300MiB of memory and 0.2 vCPU, the total overhead is 4.8GiB of memory (300MiB times 16 apps) and 3.2 vCPU (0.2 vCPU times 16 apps).

That means you're paying for resources, sometimes 6 to 10% of which you can't use.

1



2



3

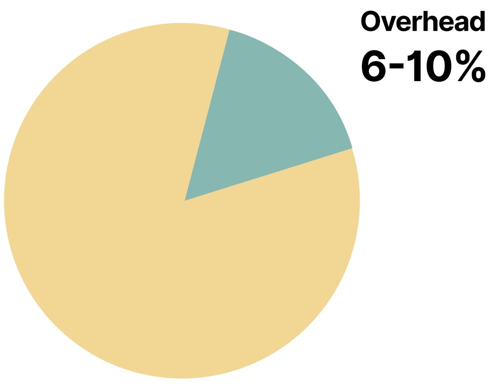


Fig. 1 If you assume that you can break each app down into four components, you should now have 16 virtual machines and operating systems.

Fig. 2 Smaller components have modest CPU and memory requirements. The CPU and memory used by the operating system aren't negligible anymore since the virtual machine is smaller too.

Fig. 3 The overhead in running operating systems with smaller virtual machines is more significant.

However, the operating system overhead is only part of the issue.

Poor resource allocation

You have probably realised that when you break your service into smaller components, each of them comes with different resource requirements for CPU and memory.

Some components, such as data processing and data mining applications, are CPU intensive.

Others, such as servers for real-time applications, might use more memory than CPU.

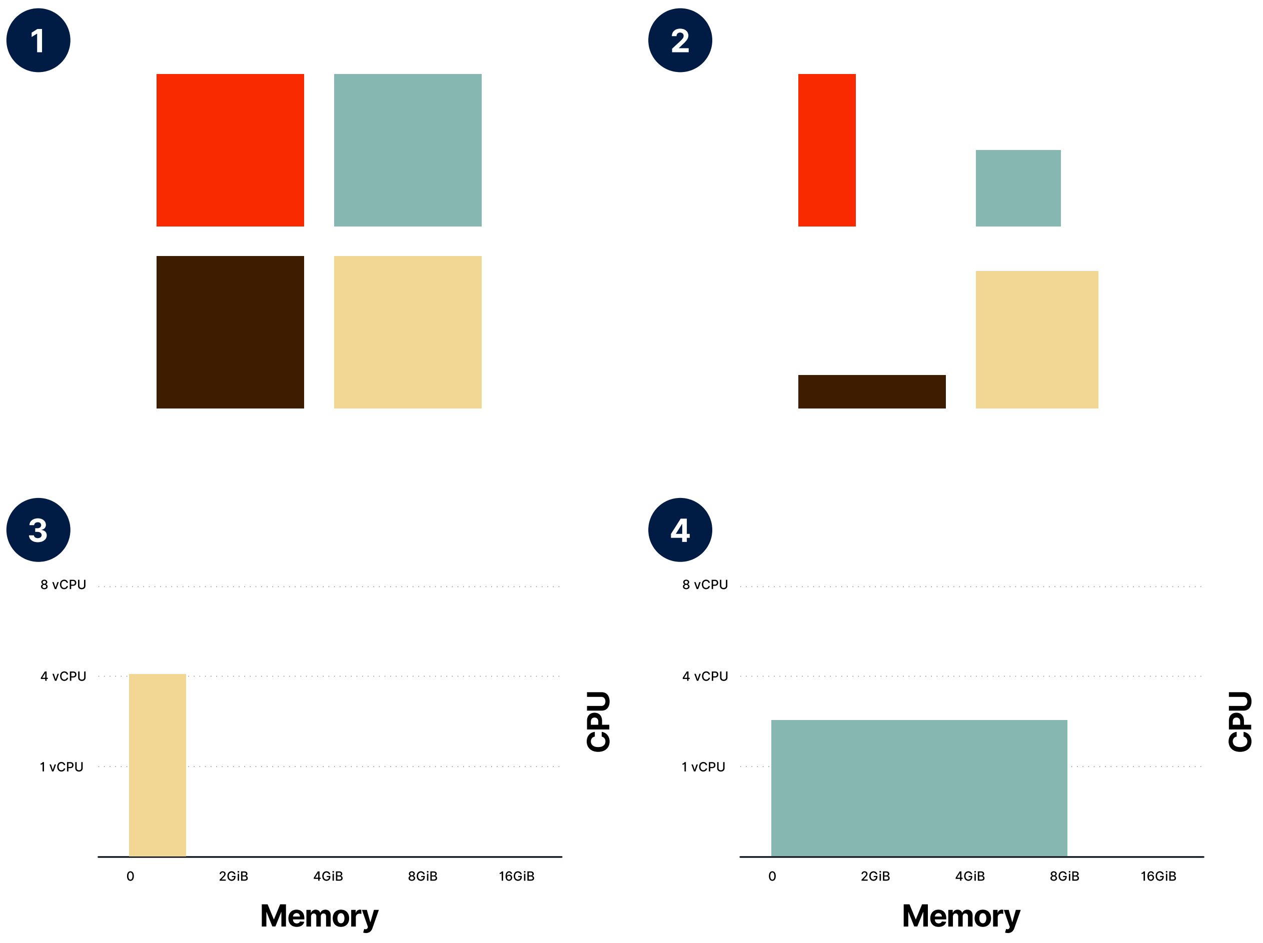


Fig. 1 When you develop apps as a collection of smaller components, you realise that no two apps are alike.

Fig. 2 They all look different. Some of them are CPU intensive; others require more memory. And you might have apps requiring specific hardware such as GPUs.

Fig. 3 You can imagine being able to profile those apps. Some of them could use more CPU than others.

Fig. 4 Or you could have components that use similar CPU resources but a lot more memory.

Ideally, you should strive to use the right virtual machine that fits your app's requirements.

If you have an application that uses 800MiB of memory, you don't want to use a virtual machine that has 2GiB.

You're probably fine with one that has 1GiB.

In practice, it's easier said than done.

It's more practical to select a single virtual machine that is good enough in 80% of the cases and use it all the time.

The result?

You waste hundreds of gigabytes of RAM and plenty of CPU cycles in underutilised hardware.

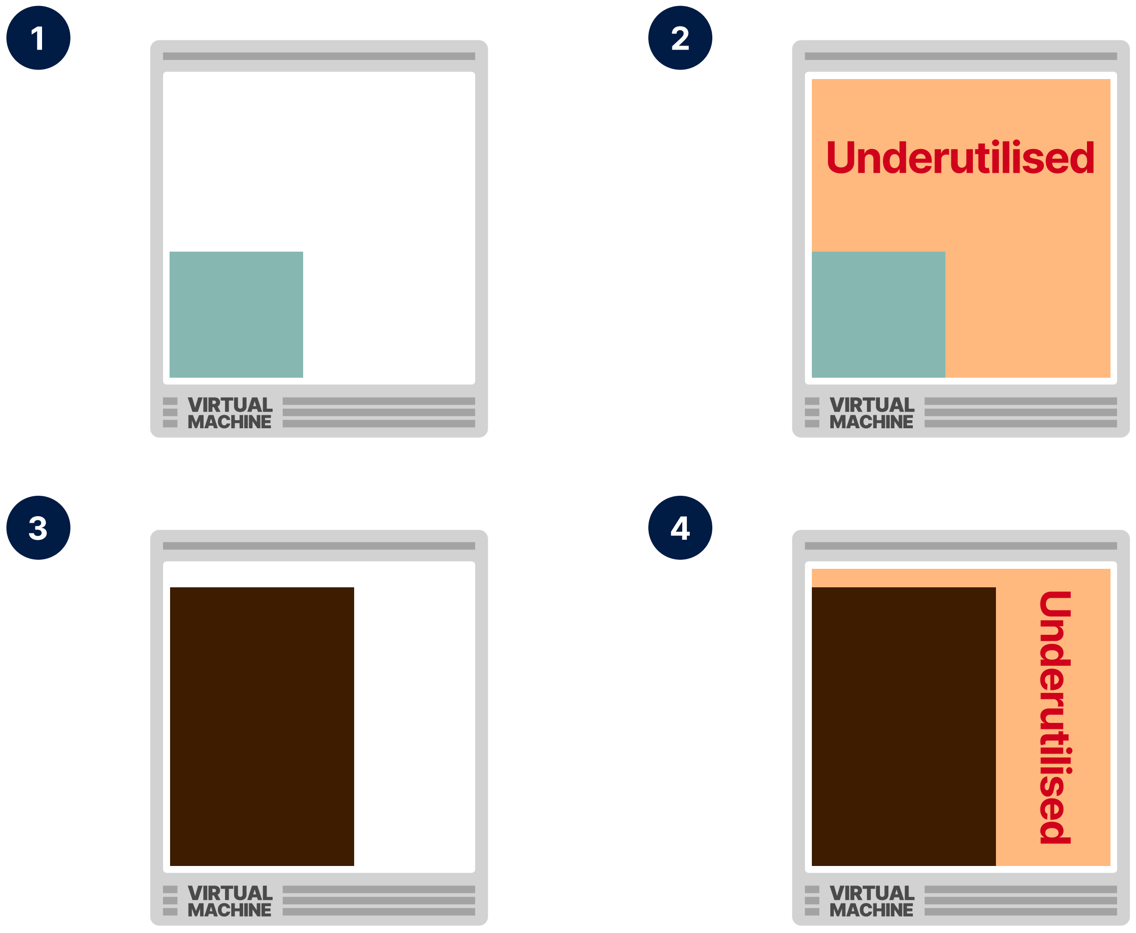


Fig. 1 You can deploy an application that uses only 1GiB of memory and 1 vCPU in a 4GB memory virtual machine.

Fig. 2 However, you're wasting 3/4 of the resources.

Fig. 3 It's common to use the same virtual machine's spec for all apps. Sometimes you might be lucky and minimise the waste in CPU and memory.

Fig. 4 In this case there are still resources left, but they are negligible.

Companies are utilising as little as 10% of their allocated resources.

Can you imagine allocating hundreds of servers, but effectively using only a dozen of them?

What you need is something to break free from the fixed resource allocation of a virtual machine and regain the resources that you don't use.

If you don't use the CPU and memory, you should be able to claim it back and use it for other workloads.

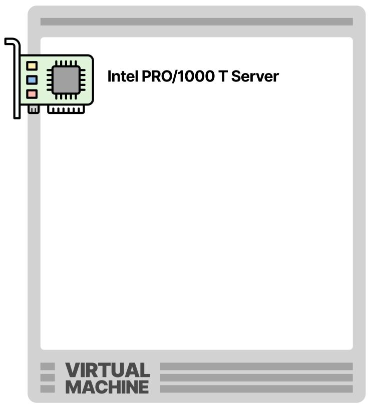
But that's not the only waste of resources.

A virtual machine emulates a real computer.

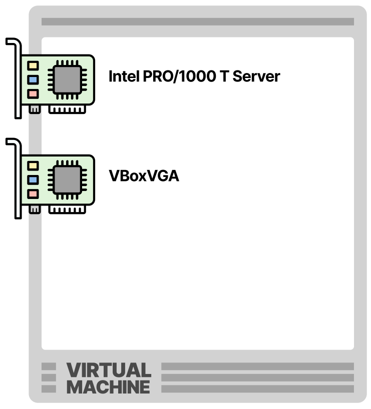
When you create one, you can decide what kind of CPU you wish to emulate, as well as what networking device to use, storage, etc.

If you have hundreds of virtual machines, each of them will have their own virtual network interface — even if some of them share the same network connection on the same server.

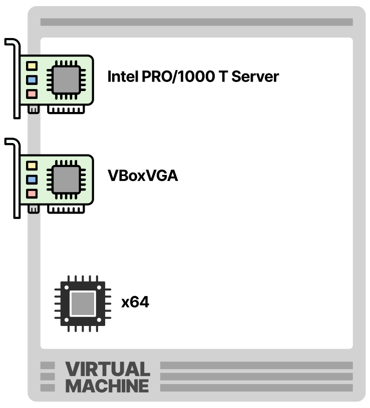
1



2



3



4

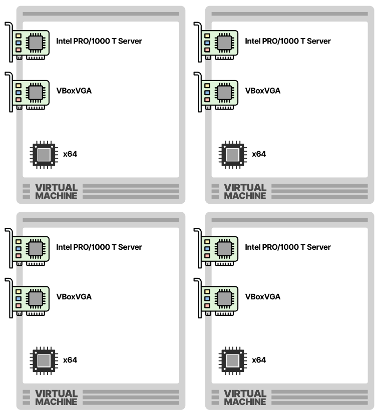


Fig. 1 Virtual machines emulate network devices to connect to the internet.

Fig. 2 They also emulate graphic cards.

Fig. 3 If you wish to do so, you could run a CPU with a different architecture from your computer.

Fig. 4 Emulating hardware is costly, particularly when you run virtual machines at scale.

The other challenge you might face is the packaging and distribution of your virtual machines.

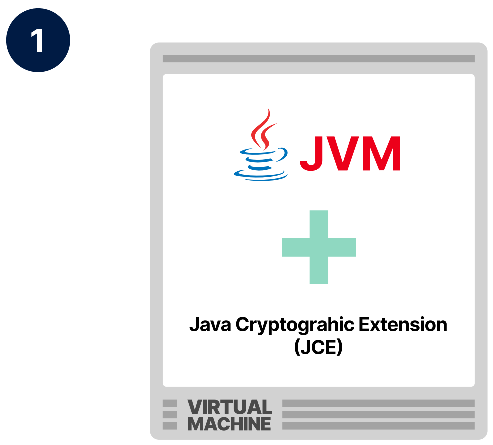
Packaging and distribution

A virtual machine is just a regular server, so you have to install, run and maintain an operating system.

You also need to provide the environment with the right dependencies.

If you develop Java apps, you might need the JVM (runtime) as well as external dependencies (a good example is the Java Cryptographic Extension which has to be installed separately).

Similarly, Node.js applications require the Node.js runtime as well as Python and the C/C++ toolchain to compile native add-ons.



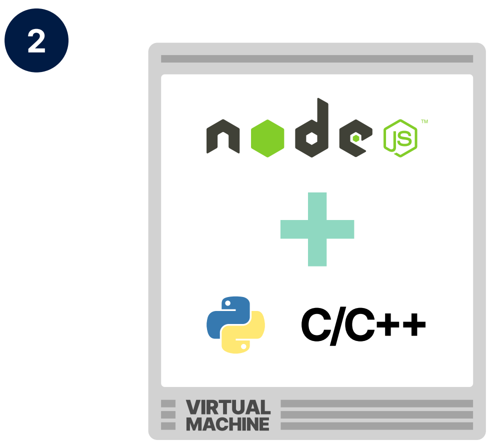


Fig. 1 If you plan on running a Spring application, you might need to provision a virtual machine with the JVM installed.

Fig. 2 If you wish to run a Node.js app, you might need to provision the environment accordingly.

You should cater to all of those requirements when you create the virtual machine and before the service is deployed.

If it sounds like a time-consuming and complex task, it is.

Fortunately, there's an entire set of tools designed to configure and provision environments such as Puppet, Ansible, Chef, Salt, etc.

There isn't a standard way to provision environment, so you can pick the tool that works best for you.

But even if you automate the configuration, it is often not enough.

Launching a virtual machine, waiting for the operating system to boot, and installing all of the dependencies could take some time.

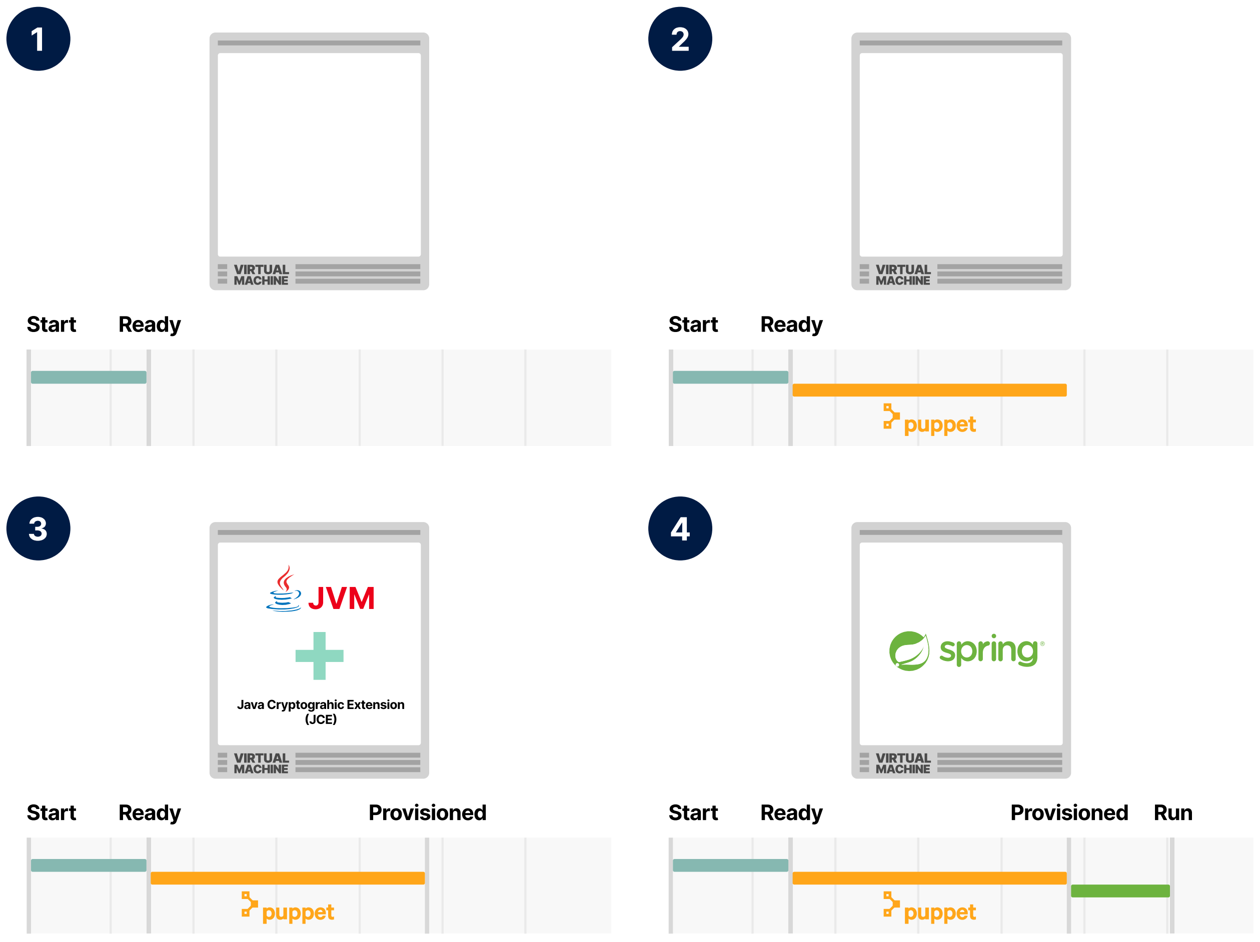


Fig. 1 Provisioning a virtual machine is time-consuming. Waiting for it to be created and ready could take a few minutes.

Fig. 2 Even if you automate the configuration, downloading and installing packages could take several minutes.

Fig. 3 At the end of the process, you still have to download the code for your app and run it.

Fig. 4 Since that could take time too, provisioning environments from scratch could take several minutes and isn't best practice.

It's also error-prone.

What if one of the packages could not be downloaded?

You have to start from scratch.

When working with virtual machines, it's usually a good idea to provision the environment once, take a snapshot and save it for later.

When you need a new environment, you can retrieve the snapshot, make a copy and deploy the application.

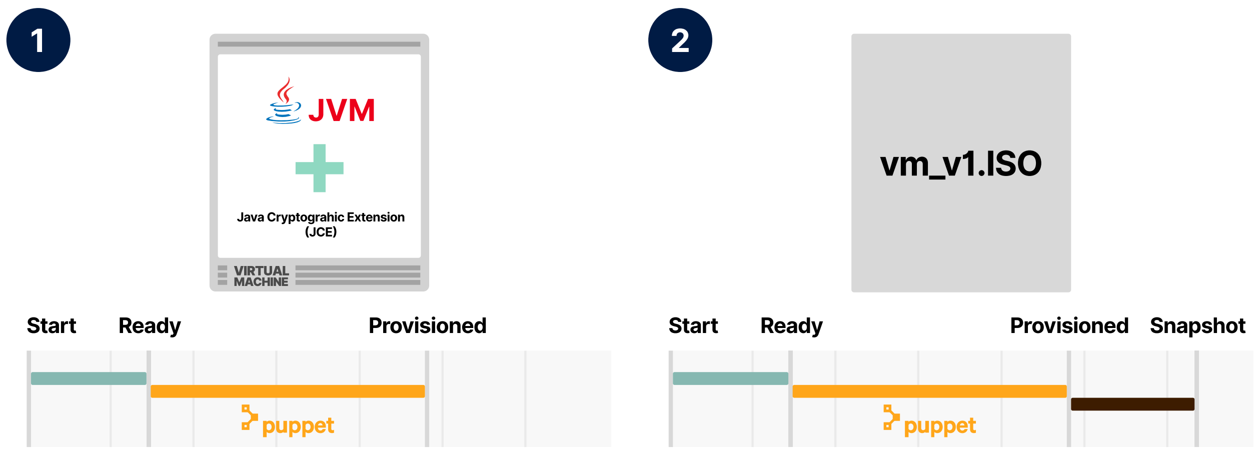


Fig. 1 Once you provision the virtual machine, you don't have to run the environment immediately.

Fig. 2 Instead, you could save the image and use it to generate copies the next time you need one.

Taking snapshots of virtual machines is a popular option since you can speed up the deployment process significantly.

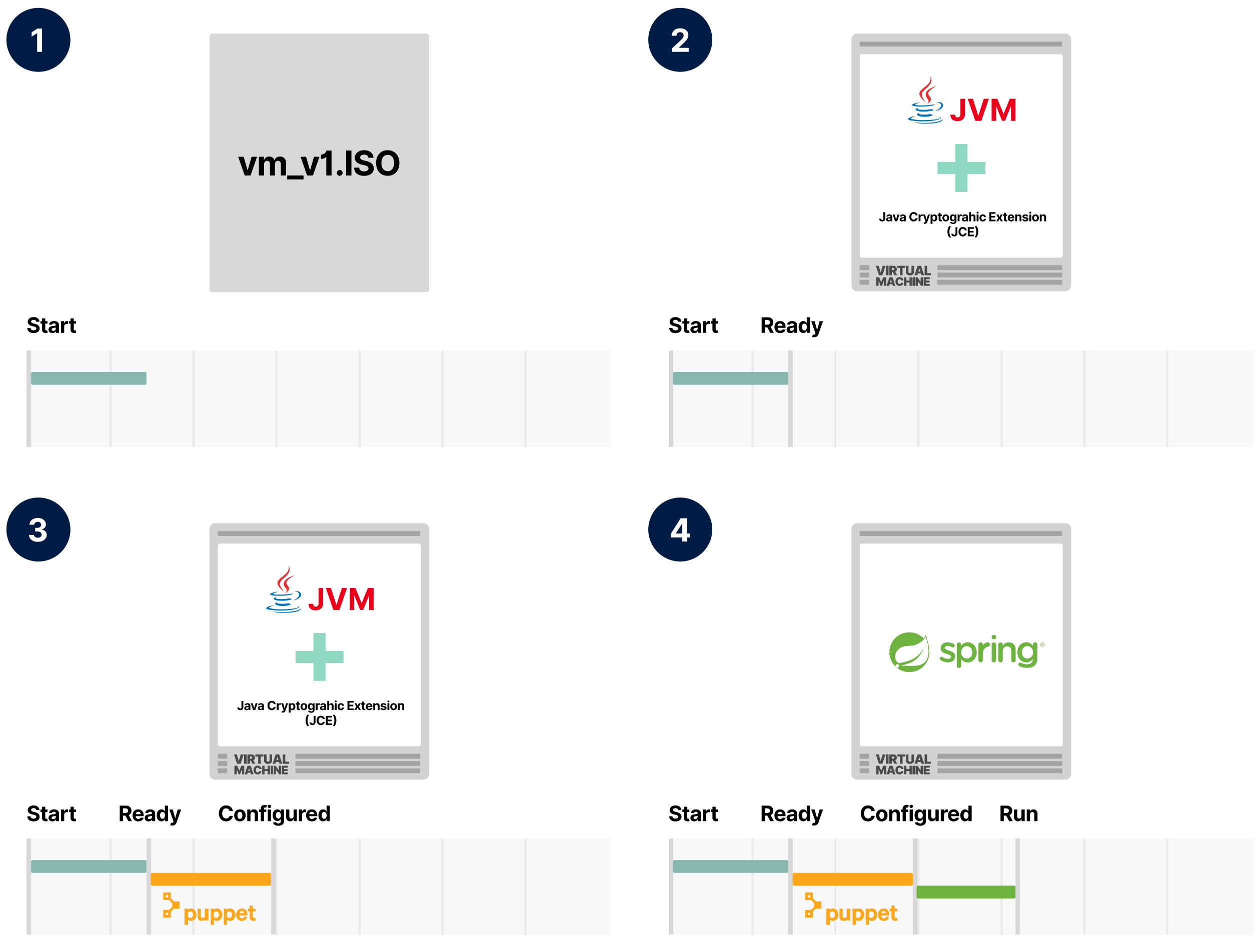


Fig. 1 When you wish to create an environment, you could copy the snapshot instead of reprovisioning a blank virtual machine.

Fig. 2 As soon as the virtual machine boots, the environment is preconfigured.

Fig. 3 You might still need to apply configurations to it, such as setting the proper environment variables.

Fig. 4 However, it takes significantly less time to create a new environment.

Unfortunately, there is no common standard for creating snapshots.

If you use open source tools such a Virtual Box, you might use the vbox format.

Amazon Web Service expects virtual machines to use the Amazon Machine Image (AMI).

Azure expects those snapshots to be packaged in a completely different format too.

While you might be able to convert from one format to another, it's not always straightforward to do so.

And you also have to version and keep track of snapshots as you patch the operating system or upgrade the dependencies.

Every change to the virtual machine has to go through a new cycle of provisioning and snapshotting.

If you deploy applications with diverse languages and runtimes, you might find yourself automating the process of creating, patching and deployments for each app.

With dozens or hundreds of applications, you can already imagine how much effort is involved in creating and maintaining such a system.

As an example, let's consider the following set-up:

- One Node.js application.
- One Java application.
- Two environments: development and production.

If you decide to adopt a production set-up based on virtual machines, you might need:

- A generic script that can provision the environment. The script should install all the dependencies before the snapshot is taken.
- A repository where you can store and tag snapshots for later retrieval.
- A script to provision environment-specific settings such as environment variables.
- A script that retrieves the latest snapshot, creates a virtual machine and deploys the app.

The above should be repeated twice — one for each app.

Every organisation has a slight variation of the above steps.

Some might skip steps like the snapshotting, for example.

Others might integrate with more advanced checks and optimisation such as scanning snapshots for common vulnerabilities.

What's important to recognise is that there isn't a widely adopted standard or common way to do things.

As a consequence, there's fragmentation in tools and most of the time the code is vendor-specific.

The automation that you build to deploy apps in Amazon Web Service cannot be reused in Azure unless you make changes.

Recap

Managing applications at scale is challenging.

Maintaining and running infrastructure for thousands of applications can be even more challenging.

Virtual machines are an excellent mechanism to isolate workloads, but they have limitations:

1. **Resources are allocated upfront.** Every CPU cycle or megabyte of memory that you don't use is lost.

2. **Each virtual machine runs an operating system that consumes memory and CPU.** The more virtual machines, the more resources you have to spend to keep operating systems running.
3. **Virtual machines emulate even the hardware — even if you don't need it.** A virtual machine is a virtual computer that emulates network drivers, CPU, storage, etc.
4. **You should invest in tooling and processes to create, run and maintain virtual machines.** The industry hasn't settled on a single tool or strategy, so there is little by way of an off the shelf solution that you can leverage.

But are virtual machines the only mechanism to isolate workloads?

In the next chapter you will learn how you can isolate apps without using virtual machines.