Extracted from:

Release It! Second Edition

Design and Deploy Production-Ready Software

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Release It!

Second Edition

Design and Deploy
Production-Ready Software

Michael T. Nygard
Edited by Katharine Dvorak

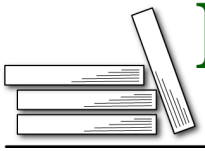
Release It! Second Edition

Design and Deploy Production-Ready Software

Michael T. Nygard

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-239-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Physical Hosts, Virtual Machines, and Containers

At some level, all machines are the same. Eventually, all our software runs on some piece of precisely patterned silicon. All our data winds up on glass platters of spinning rust or encoded in minute charges on NAND gates. That's where the similarity ends. A bewildering array of deployment options force us to think about the machines' identities and lifespans. These aren't just packaging issues, either. A design that works nicely in a physical data center environment may cost too much or fail utterly in a containerized cloud environment. In this section, we'll look at these deployment options and how they affect software architecture and design for each kind of environment.

Physical Hosts

The CPU is one place where the data center and the development boxes have converged. Pretty much everything these days runs a multicore Intel or AMD x86 processor running in 64-bit mode. Clock speeds are pretty much the same, too. If anything, development machines tend to be a bit beefier than the average pizza box in the data center these days. That's because the story in the data center is all about expendable hardware.

This is a *huge* shift from just ten years ago. Before the complete victory of commodity pricing and web scale, data center hardware was built for high reliability of the individual box. Our philosophy now is to load-balance services across enough hosts that the loss of a single host is not catastrophic. In that environment, you want each host to be as cheap as possible.

There are two exceptions to this rule. Some workloads require large amounts of RAM in the box. Think "graph processing" rather than ordinary HTTP request/response applications. The other specialized workload is GPU computing. Some algorithms are "embarrassingly parallel," so it makes sense to run them across thousands of vector-processing cores.

Data center storage still comes in a bewildering variety of forms and sizes. Most of the useful storage won't be directly on the individual hosts. In fact, your development machine probably has more storage than one of your data center hosts will have. The typical data center host has enough storage to hold a bunch of virtual machine images and offer some fast local persistent space. Most of the bulk space will be available either as SAN or NAS. Don't be fooled by the similarity in those acronyms. Bloody trench wars have been fought between the two camps. (It's easier to make trenches in a data center than you might think. Just pop up a few raised floor panels.) To an application running on the host, though, both of them just look like another mount point or drive letter.

Your application doesn't need to care too much about what protocol the storage speaks. Just measure the throughput to see what you're dealing with. Bonnie 64 will give you a reasonable view with a minimum of fuss.[1]

All in all, the picture is much simpler today than it once was. Design for production hardware for most applications just means building to scale horizontally. Look out for those specialized workloads and shift them to their own boxes. For the most part, however, our applications won't be running directly on the hardware. The virtualization wave of the early 2000s left no box behind.

Virtual Machines in the Data Center

Virtualization promised developers a common hardware appearance across the bewildering array of physical configurations in the data center. It promised data center managers that it would rein in "server sprawl" and pack all those extra web servers running at 5 percent utilization into a high-density, high-utilization, easily managed whole. Guess which story turned out to be more compelling?

On the down side, performance is much less predictable. Many virtual machines can reside on the same physical hosts. It's rare to see VMs move from one host to another, because it's disruptive to the guest. (The "host operating system" is the one that really runs on hardware. It provides the virtualization features. "Guest operating systems" run in the virtual machines.) Physical hosts are usually oversubscribed. That means the physical host may have 16 cores, but the total number of cores allocated to VMs on the host is 32. That host would be 200 percent subscribed or 100 percent *over*subscribed. If all those applications receive requests at the same time, just through random chance, then there's not enough CPU to go around.

Almost any resource on the host can be oversubscribed, especially CPU, RAM, and network. Regardless of resource, the result is always the same: contention among VMs and random slowdowns for all. It's virtually impossible for the guest OS to monitor for this.

When designing applications to run in virtual machines (meaning pretty much *all* applications today) you must make sure that they're not sensitive to the loss or slowdown of any one host. That's just a good idea anyway, but it's particularly important here. Here are some things to watch out for:

- Distributed programming techniques that require synchronous responses from the whole cluster for work to proceed

1. https://sourceforge.net/projects/bonnie64

- "Special" machines like cluster managers or lock managers, unless another machine can take over without reconfiguration
- Subtle dependency on request or event ordering—nobody designs this into a system, but it can creep in unexpectedly.

Virtual machines make all the problems with clocks much worse. Most programmers carry a mental model of the clock as being monotonic and sequential. That is, a program that samples the system clock may get the same value twice but it'll never get a value less than a prior response. It turns out that's not even true for a clock on a physical machine. But on a virtual machine it can be much worse. Between two calls to examine the clock, the virtual machine can be suspended for an indefinite span of real time. It might even be migrated to a different physical host that has a clock skew relative to the original host. A clock on a virtual machine is not necessarily monotonic *or* sequential. The virtualization tools try to paper over this with a little communication from the VM to query the host so the VM can update its OS clock whenever it wakes up. That keeps the VM's OS clock synced with the host's OS clock. From an application perspective, this makes the clock jump around even more. The bottom line is: don't trust the OS clock. If external, human time is important, use an external source like a local NTP server.

Containers in the Data Center

Containers have invaded the data center, pushed there by developer insistence. Containers promise to deliver the process isolation and packaging of a virtual machine together with a developer-friendly build process. The container hypothesis says, "I'll never again have to ask if production matches QA."

Containers in the data center act a lot like virtual machines in the cloud (see *Virtual Machines in the Cloud*, on page ?). Any individual container only has a short-lived identity. As a result, it should not be configured on a per-instance basis. This can cause interesting effects with older monitoring systems (looking at you, Nagios!) that need to be reconfigured and bounced every time a machine is added or removed.

A container won't have much, if any, local storage, so the application must rely on external storage for files, data, and maybe even cache.

The most challenging part of running containers in the data center is definitely the network. By default, a container doesn't expose any of its ports (on its own virtual interface) on the host machine. You can selectively forward ports from the container to the host, but then you still have to connect them from one host to another. One common pattern that's developing is the *overlay network*. This

uses virtual LANs (VLANs)—see *Virtual LANs for Virtual Machines*, on page 9—to create a virtual network just among the containers. The overlay network has its own IP address space and does its own routing with software switches running on the hosts. Within the overlay network, some control plane software manages the whole ensemble of containers, VLANs, IPs, and names.

A close second for "hardest problem in container-world" is making sure enough container instances of the right types are on the right machines. Containers are meant to come and go—part of their appeal is their very fast startup time (think milliseconds rather than minutes). But that means container instances will be like quantum foam burbling across all your hosts. Manually operating containers would be absurd. Instead, we delegate that job to another bit of control plane software. We describe our desired load out of the containers, and the software spreads container meringue across the physical hosts. The control software should know something about the geographic distribution of the hosts as well. That way it can allocate instances regionally for low latency while maintaining availability in case you lose a data center.

It seems natural that the same software should schedule container instances and manage their network settings, right? Solutions for running containers in data centers are emerging. None are dominant at this time, but packages like Kubernetes, Mesos, and Docker Swarm are attacking both the networking and allocation problem. Whichever one solves this problem first will be able to truly claim the title of "operating system for the data center."

When you design an application for containers, keep a few things in mind. First, the whole container image moves from environment to environment, so the image can't hold things like production database credentials. Credentials all have to be supplied to the container. A 12-factor app handles this naturally. If you're not using that style, think about injecting configuration when starting the container. In either case, look into password vaulting.

The second thing to externalize is networking. Container images should not contain hostnames or port numbers. Again, that's because the setting needs to change dynamically while the container image stays the same. Links between containers are all established by the control plane when starting them up.

Virtual LANs for Virtual Machines

As if there weren't enough ways for a packet to hit a pocket on a socket on a port, we've got virtual LANs (VLANs) and virtual *extensible* LANs (VXLANs) to contend with. The idea of a VLAN is to multiplex Ethernet frames on a single wire but let the switch treat them like they came in from totally separate networks. The VLAN tag is a number from 1 to 4,094 that nestles into the physical routing portion of the header. Every network you encounter will support VLANs.

The operating system that runs a NIC can create a virtual device assigned to a virtual LAN. Then all the packets sent by that device will have that VLAN ID in them. That also means the virtual device must have its own IP address in a subnet assigned to that VLAN.

VXLAN takes the same idea but runs it at "layer 3," meaning it's visible to IP on the host. It also uses 24 more bits in the IP header, so a physical network can have more than 16 million VXLANs riding its wires.

At one time this was all the province of network engineers pulling cables around the data center. Virtualization and containers increasingly rely on software switches to handle dynamic updates. It will be common to see software switches running on the hosts, presenting a complete network environment to the containers that does the following:

- Allows containers to "believe" they're on isolated networks
- Supports load-balancing via virtual IPs
- Uses a firewall as a gateway to the external network

While this technology matures, our container systems have to provide their own load-balancing and need to be told which IP addresses and ports their peers are on.

The 12-Factor App

Originally created by engineers at Heroku, the 12-factor app is a succinct description of a cloud-native, scalable, deployable application.[a] Even if you're not running in a cloud, it makes a great checklist for application developers.

The "factors" identify different potential impediments to deployment, with recommended solutions for each:

Codebase
: Track one codebase in revision control. Deploy the same build to every environment.

Dependencies
: Explicitly declare and isolate dependencies.

Config
: Store config in the environment.

Backing services
Treat backing services as attached resources.

Build, release, run
Strictly separate build and run stages.

Processes
Execute the app as one or more stateless processes.

Port binding
Export services via port binding.

Concurrency
Scale out via the process model.

Disposability
Maximize robustness with fast startup and graceful shutdown.

Dev/prod parity
Keep development, staging, and production as similar as possible.

Logs
Treat logs as event streams.

Admin processes
Run admin/management tasks as one-off processes.

See the website for greater detail on each of these recommendations.

a. https://12factor.net

Containers are meant to start and stop rapidly. Avoid long startup or initialization sequences. Some production servers take many minutes to load reference data or to warm up caches. These are not suited for containers. Aim for a total startup time of one second.

Finally, it's notoriously hard to debug an application running inside a container. Just getting access to log files can be a challenge. Don't even bother trying to figure out why some socket is being held open for too long. Containerized applications, even more than ordinary ones, need to send their telemetry out to a data collector.