CS-310 Scalable Software Architectures

Lecture 18:
Computing Platforms

Steve Tarzia

Last Time: Twitter Database Architecture

- Twitter's storage design choices offer a tradeoff between:
  - Relational DB: space-efficient, fast writes, but slow reads.
  - NoSQL DB: duplicative, slow writes, but fast reads.
- A hybrid design is ideal:
  - Most users are **consumers** (reads > writes): put their tweets in NoSQL.
  - **Celebrities** are different (writes > reads): put their tweets in SQL.

App **packaging, distribution,** and **deployment** is tricky

- Unfortunately, an app usually requires a very specific **environment**:
  - Application server or middleware. Eg., Tomcat, Maven, Express.js
  - Language interpreter / runtime. Eg., Python 3.5, Java 10, Node.js
  - Libraries *(within the language)*. maybe loaded by mvn, npm, pip, ld
  - Command line tools & services. Eg., imagemagick, mysql, openssl
  - Operating system features & config. Eg., filesystem structure, firewall, subprocesses/threads, user security.
- As on the homework, *"it worked on my machine"* is not good enough.
- Many app packaging solutions exist, and they are constantly evolving.
  - We'll compare some common choices:
    Java **JVM**, Full **VMs**, **Containers**, **Serverless functions**

Your experiences?

- What have been your best/worst experiences trying to run code someone else has written?
- What have been your best/worst experiences distributing code you've written?

- What language/tool features make distribution easier?

Packaging option #0: **Binary Executable**

- Common for distributing apps on consumer OSes.
- Windows and Mac **desktop** apps are distributed through an *app store* or a special *installer* program.
    - Both app stores and installers take care of installing prerequisite libraries.
    - Or distribute a **binary executable**. Will only run on very similar systems.
    - *Windows* has been carefully designed for backward compatibility.
        - Apps packaged for Windows XP (2001) should still run on Windows 10 today!
- Android and iOS **mobile** apps are easier to package because they are more heavily *sandboxed*. Cannot access outside command line or files.
    - Anyway, developers use tools provided by the OS for packaging apps.
- Linux *distributions* have package managers like **apt** and **yum** to install FOSS program binaries and their dependencies.

Packaging option #1: **Source code + README**

- Most small code projects are not well packaged.
- Post the source code somewhere (github!).
- Maybe write a README with some instructions for how to set up the environment.
- Hopefully provide a build script (makefile, requirements.txt, rakefile, build.sh, *depending on the language*).

✗ Low probability that user will be able to reproduce a working env.

It worked on my machine…

- Exception: Maven/Gradle for Java source code is pretty reproducible.

Packaging option #2: **JVM Bytecode**

- Compiled Java code runs on **Java Virtual Machine** (JVM).
  - JRE *emulates* the JVM and translates "hot" (critical) JVM code to native code.
- "Write once, run anywhere" (hopefully).
- Code is distributed as .jar or .war file *(Java bytecode archive)*
- Many new languages also compile to the same JVM:
  - Write code in Java *or* Scala, Kotlin, Clojure, Groovy, Jython, Jruby.

Pros: ✓ Small code size. ✓ Simplicity. ✓ Performance.

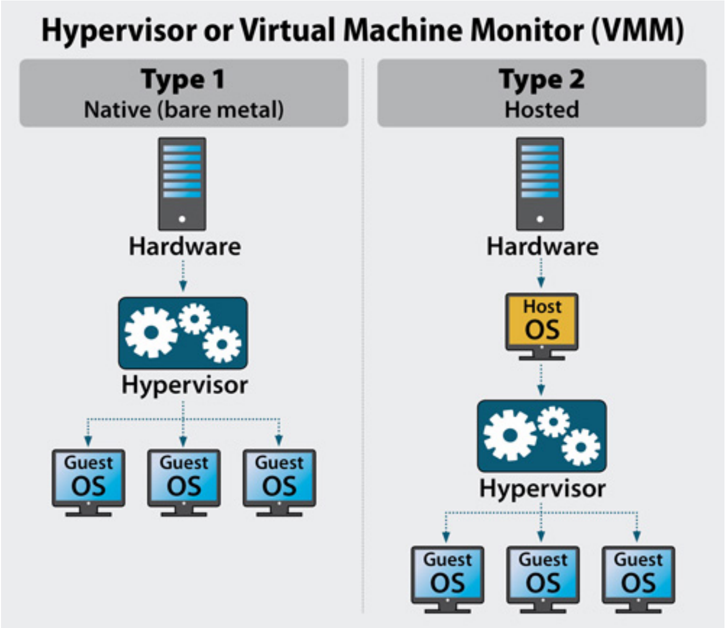
Cons: ✗ Limited choice of languages.

✗ App is not truly isolated from host OS.

✗ No control of outside environment (eg., command-line tools and filesystem).

Virtual Machines

- In the old days, you installed **one Operating System** on a server.
  - To share resources (and save money), you might run both a web server and an email server simultaneously on the same OS/machine.
  - Think of your own laptop. The OS can share one CPU/memory among many concurrent applications (called processes). *(details in CS-343)*
  - Nowadays, running multiple server apps on one OS is rare.
- A **hypervisor** or VMM allows a single machine run multiple **virtual machines**.
  - Each VM has its own OS installed, can be the same OS version or different. VMs are isolated. VMs can communicate over the network.
  - Hypervisors include: KVM, Xen, VMware, Hyper-V.



Why use virtual machines?

The hypervisor introduces complexity and some inefficiency, but…

- VMs allow problems in one service to be isolated from other services.
  - Eg., if *web* server has a [memory leak | security breach | OS crash], the *mail* server on another VM will not be affected.
- VMs can be migrated on the fly *(if supported by the hypervisor)*.
  - If physical machine must be retired or replaced, the guest VMs can continue running on another machine.
- Servers are big, but you may not require a full machine.
  - Eg., *moore* has 48 CPU cores and 256 GB or RAM.
  - Eg., *stevetarzia.com* webserver needs < one CPU core and 512mb of RAM.
- Resources allocated to VMs can be dynamically scaled *vertically*.
  - Hypervisor can grant more RAM or CPU cores to a VM, just reboot to see it!

Elastic Compute Cloud (EC2)

- EC2 is AWS' virtual machine rental service, started in 2006.
- Outsourcing server mgmt is an old idea. Amazon's innovation was to charge by the **hour**, not month, and it started a cloud computing revolution.
- Customer chooses:
  - VM "size" (# CPU cores, RAM, network bandwidth, GPU?).
  - Operating system (various versions of Linux, Windows, now even Mac).
  - Storage type/size (SSD or magnetic? what capacity? billed separately).
  - Location. us-east-1 (Virginia), us-east-2 (Ohio), ap-east-1 (Hong Kong), …
  - Reservation period (no contract, one year, or three year).
  - Networking details (on the public Internet, or in a private network?)
  - Login credentials (SSH public key, so you can log in after it's created!)
- A VM **instance** is created for you within a few minutes.
- Billed by the hour ($0.003 to $26.68 per hour = $2 to $19,200/month)

VM operational concerns

Renting a VM frees you from *hardware* concerns, but many software operational concerns remain. These are all called **DevOps**:

- Install and configure 3rd party software:
  - databases, web servers, libraries, distributed caches, message queues, coordination tools, OS updates, etc., etc.
- Deploy new versions of your application when released.
- Monitor application and OS health:
  - Log files, CPU utilization, free memory, free disk space.
- Manage security:
  - Configure users, set/rotate passwords/keys, configure firewall.

Packaging option #3: **Virtual Machine**

- An extreme option for packaging an app.
- Ship the entire software stack, from boot code onward.
- Any hypervisor using the same CPU architecture (*eg.*, *x86-64*) and emulating same generic hardware (*network card, disk controller*) can run the code.
- Create a virtual hard disk big enough to install the OS, libs, & app.
  - At least a Gbyte or so. (maybe 1,000× larger than your source code).
- On AWS, pre-configured VMs are called **AMI**s (Amazon machine images).

Pros: ✓ Consistent & controlled runtime environment.

✓ Isolates app from outside apps.

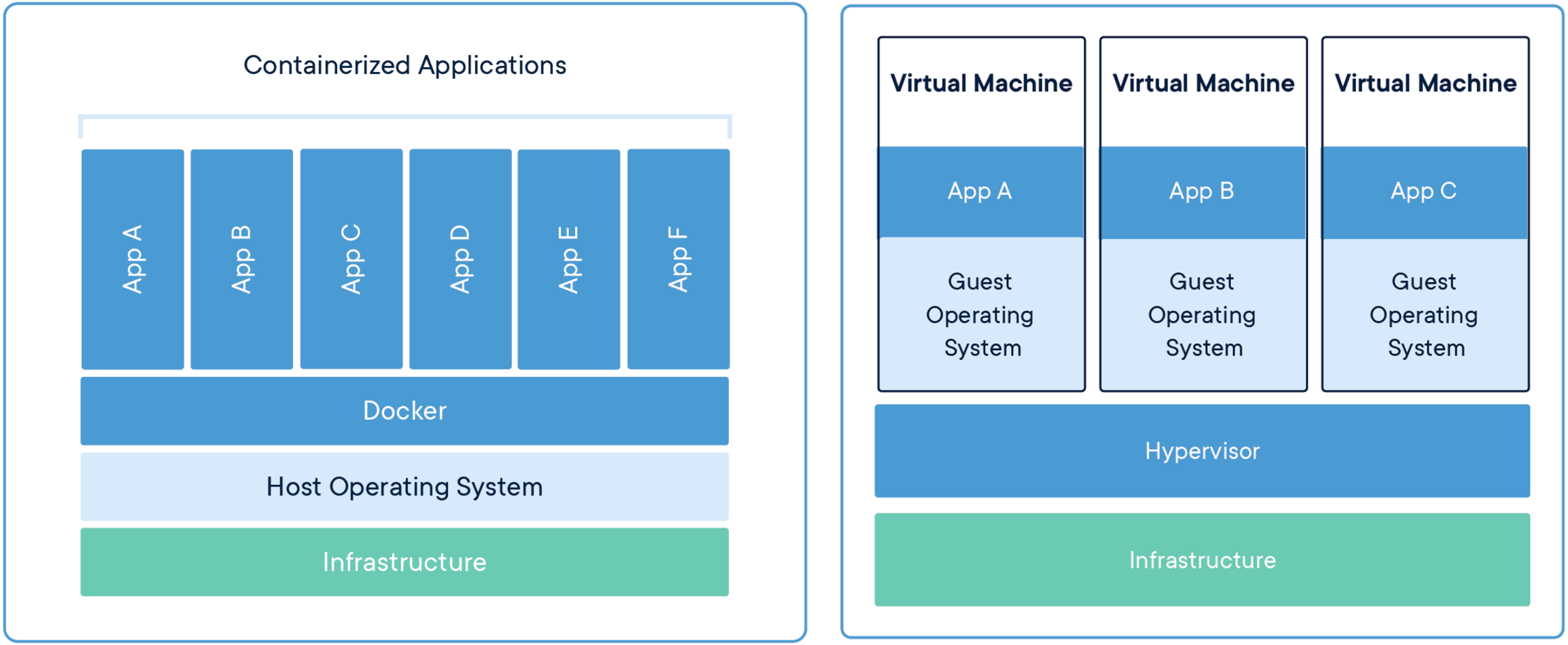
Cons: ✗ Large in size. ✗ Performance overhead for virtual/guest OS.

✗ Slow startup (must boot the OS before running your app).

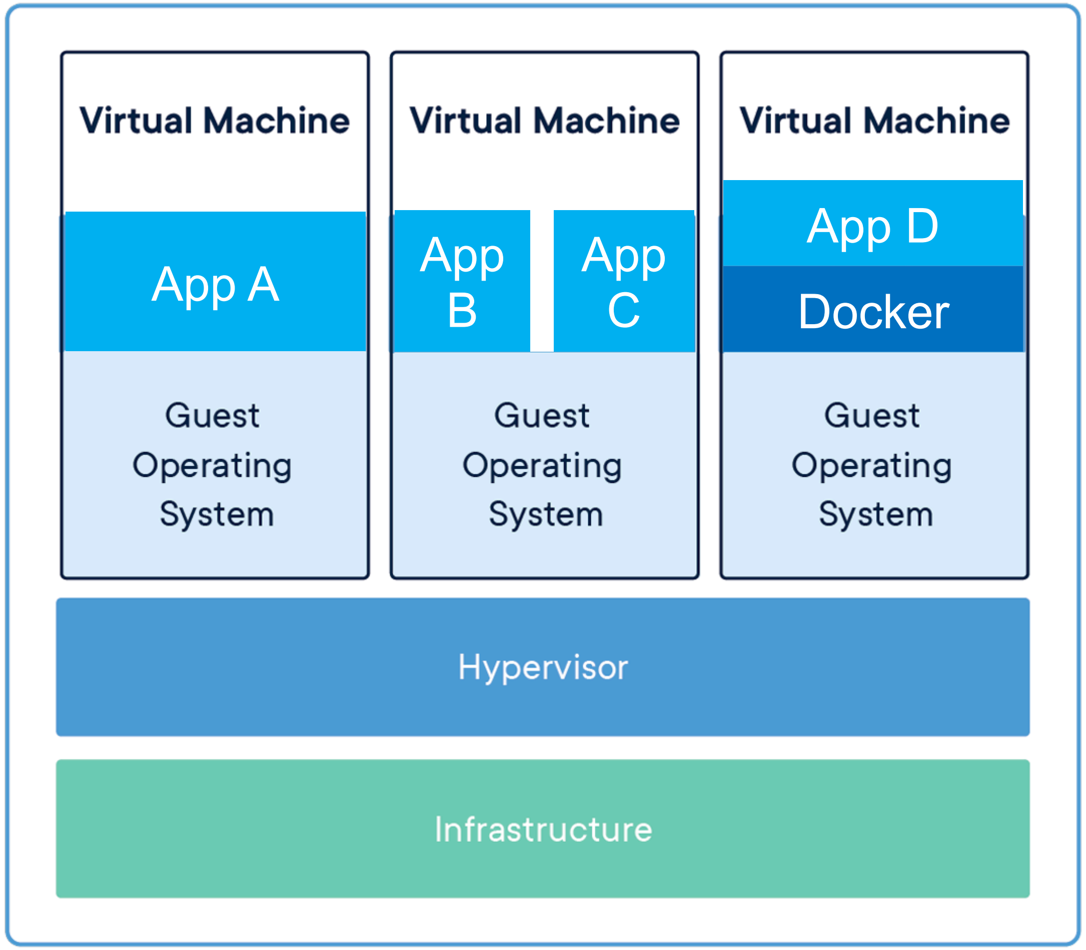
Packaging option #4: **Docker Containers**

- Similar to a VM, but much smaller and a bit more efficient.
  - Unlike a VM, a container is intended to run just one process/app.
- A *Hypervisor* runs *VMs*, *Docker* runs *Containers*.
- Container is defined by a Dockerfile which lists:
  - The **parent image** it's based on. This might be an **official base image** like alpine 3.10.3 (a tiny Linux distro). Base images are very similar to VMs.
    - Base images are very stripped-down: Alpine is 5mb, Ubuntu is 190mb.
    - Child images can add additional OS software packages as needed. (eg., "apt-get gcc")
  - Instructions to modify the parent image. Eg., this postgres container is based on the alpine image, but it downloads and configures the postgres database.
  - The dockerfile is just a small text file, but it is a **blueprint** for the app's complete runtime environment.
  - This idea borrows from OS config management tools like Puppet, Chef, Salt.

Containers *vs* Virtual Machines

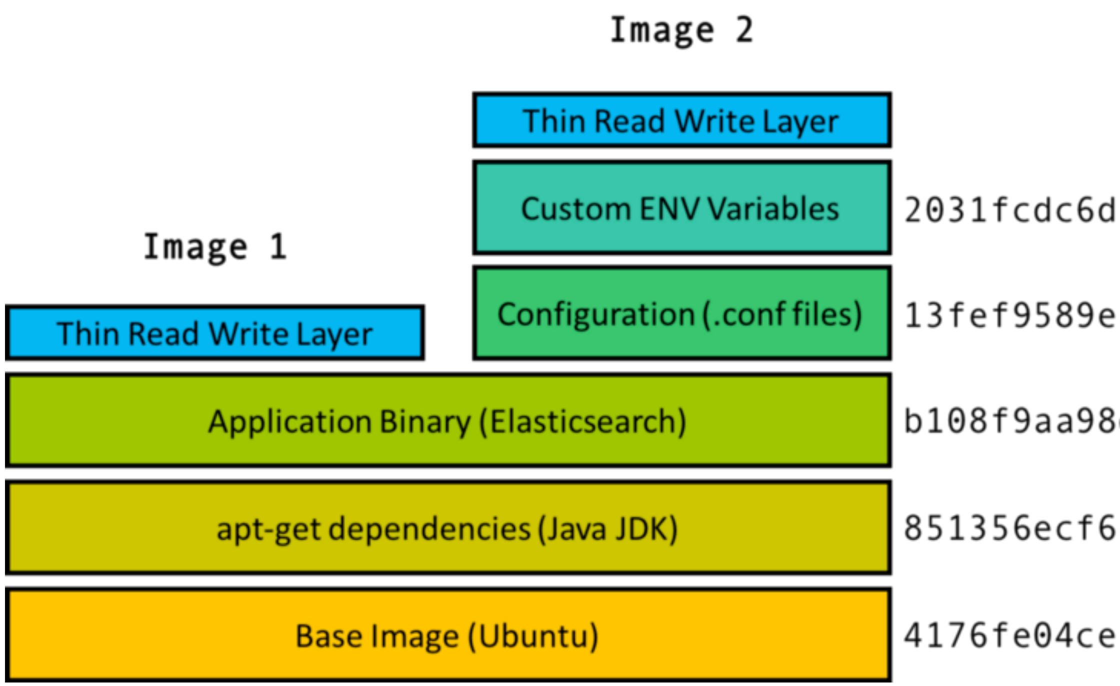


Containers can be run on VMs



Docker layered filesystem

- Recall that containers are derived from a base image.
- You might have many containers running on a machine based on the same big parent image. There is an opportunity to *share*.
- A Docker filesystem **layer** is a set of changes to a base filesystem:
- It's a *copy-on-write* filesystem.
- The figure at right shows two containers running Elasticsearch with different configurations.
- The majority of the container data is shared (bottom three layers).



Linux support for containers

- Linux has built-in support for isolating processes from each other.
- Docker uses this Linux container functionality (LXC, runc), and adds:
  - Build toolchain (Dockerfiles, etc.)
  - Layered filesystem images.
  - Central image repository (Docker hub)
  - Mac and Windows runtime environments (that run Linux in a VM).
- Note that containers cannot *see* each other, but they *feel* each other's effects more than VMs (eg., memory & CPUs are shared dynamically).

References:

- https://docs.docker.com/engine/faq/
- https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine

Where to deploy containers?

AWS gives two basic options for deploying containerized apps:

- A Virtual Machine (*EC2*) cluster.
    - **Kubernetes** is the standard *container orchestration* system for such clusters.
    - Eg., create five m4.xlarge instances running Linux & manage w/Kubernetes.
    - ✓ You can SSH into the VMs if you wish, for debugging.
    - ✗ You control # of VMs. *Idle instance?* you waste money. *Overloaded?* it's slow.
- Managed infrastructure (AWS **Fargate**, Azure **Container Instances**).
    - Your container is deployed *somewhere* (the cloud provider handles this).
    - Most likely deployed to a dedicated VM pulled from a pool, but who knows?
    - Container specifies #CPU cores and RAM size to reserve for the container.
    - ✗ You cannot SSH into it the VM and debug or configure it.
    - ✓ You pay only when your code is running, and scaling is much more dynamic.

Launch comparison

Virtual Machine

1. Copy disk image to VMM.
2. Reserve CPU and RAM share.
3. Boot VM OS.

Container

1. Copy Dockerfile and application code to host.
2. Download base image
   - Skip this step if another container is using the same base image.
3. Run setup commands in Dockerfile.
4. Run the app command in the Dockerfile.

Serverless functions abstract away the platform

- Your code just runs *somewhere* in the cloud.
- Serverless functions are heavily **constrained.** They must run on a standardized platform that can simultaneously support all tenants.
  - All code is in *one language*, on a certain runtime *(eg., Python 3.5)*.
    - Simple **Lambda** functions can actually be written/pasted in the AWS web console.
  - Dependencies must be included in the app bundle that is deployed.
    - In other words, the only thing "installed" is your function code.
    - Eg., in Python, install libs in a subdirectory: `"pip install --target ./package urllib3"`
  - Serverless functions are *stateless* (no global variables to retain state).
  - Code cannot spawn other command-line processes.
  - Code can only write to one folder in the filesystem (eg., `"/tmp"`).
    - This folder is isolated (sandboxed) from other functions running on the VM.

✓ Much less overhead and startup delay than container or VM.

👍 Note that a *Java jar* is an ideal package for a serverless function.

How Serverless Works

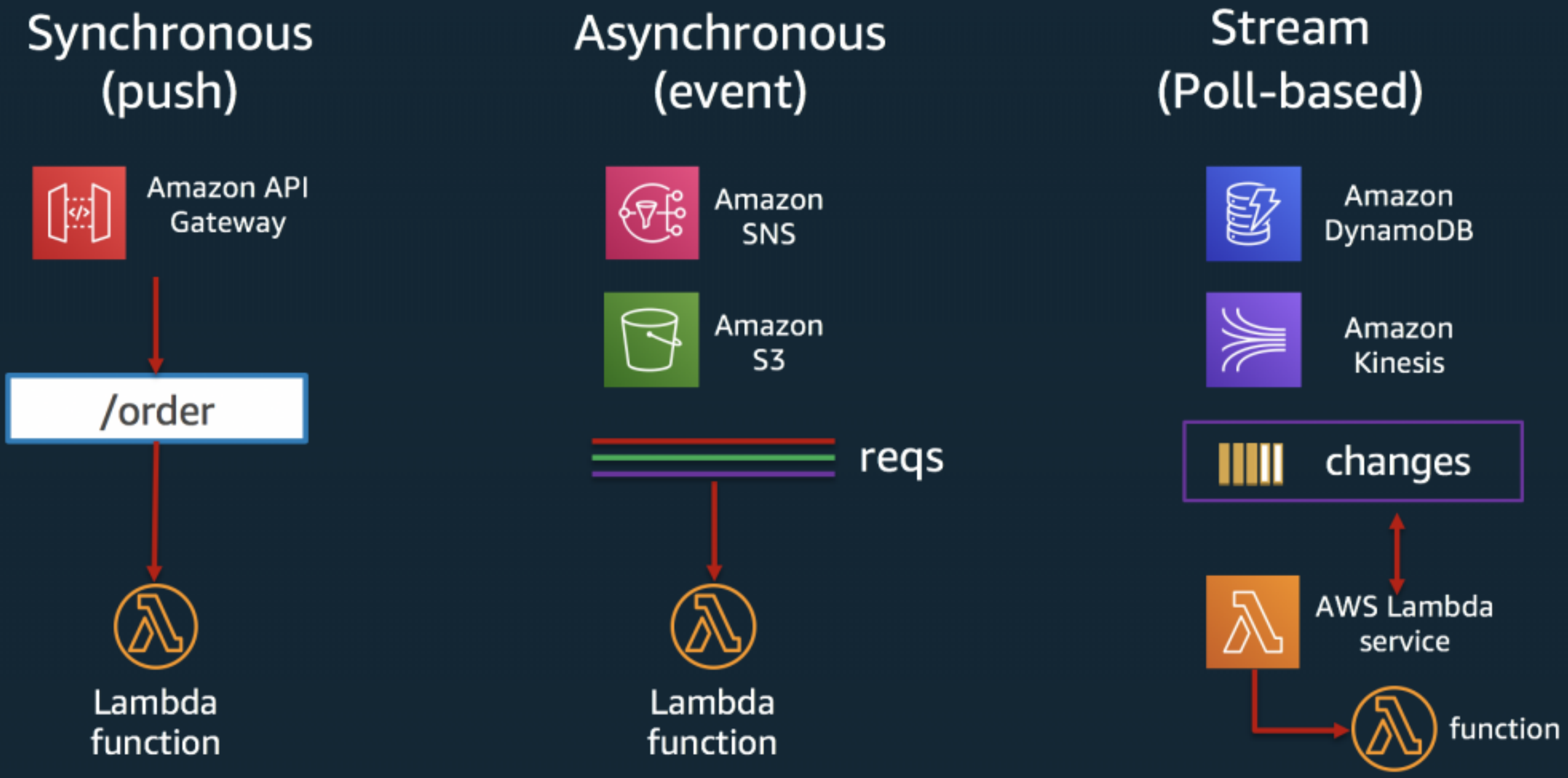
- Some cloud-configured event triggers/calls the function.
- In response, the code is deployed to a VM, perhaps along with other functions (those details are abstracted away).
- There can be a short delay in starting the function (if it's not already on a VM) called a **cold start**.
- Function is deployed to as many machines as necessary, on demand.
- AWS Lambda functions are currently limited to 15 minutes.
- The association between the code and VM is short-term.

- Note that serverless functions must be coded to work with a particular vendor's platform (eg., AWS Lamba), so there is **vendor lock-in**.

Invoking a Serverless Function

On AWS, a Lambda function can be triggered by:

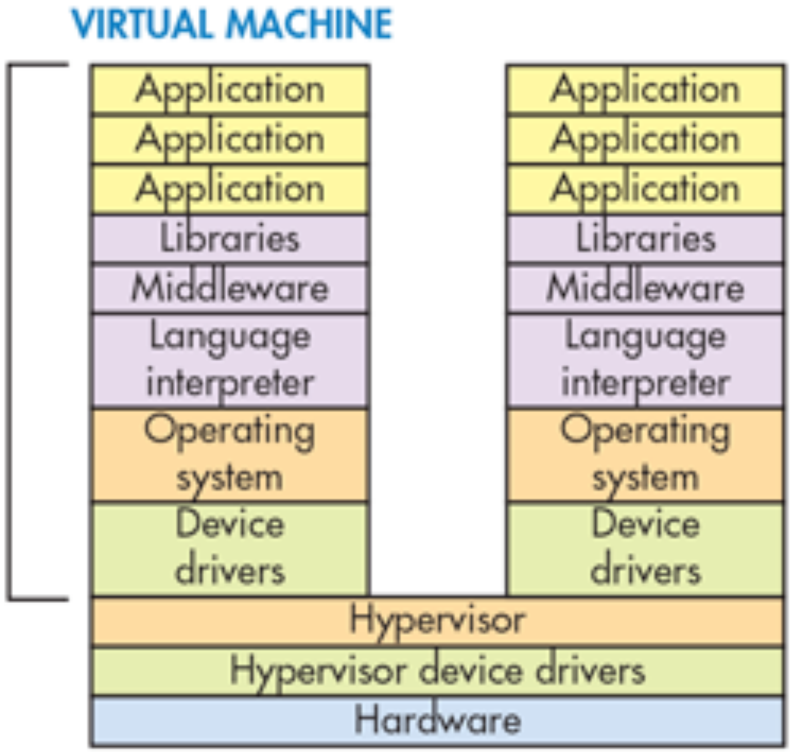
- **Admin** clicking a button in the Lambda console.
- **Your code** calling into the lambda API.
- **Client request** reaching an AWS API Gateway configured to route certain endpoints (paths) to your lambda.
- **Cloudwatch event** configured in the AWS console to run periodically.
- An event in a managed service, like S3 or Simple Queueing Service.
    - **S3** can be configured to run a lambda whenever a file is stored in a bucket.
    - **SQS** can run a lambda to handle messages added to the queue.

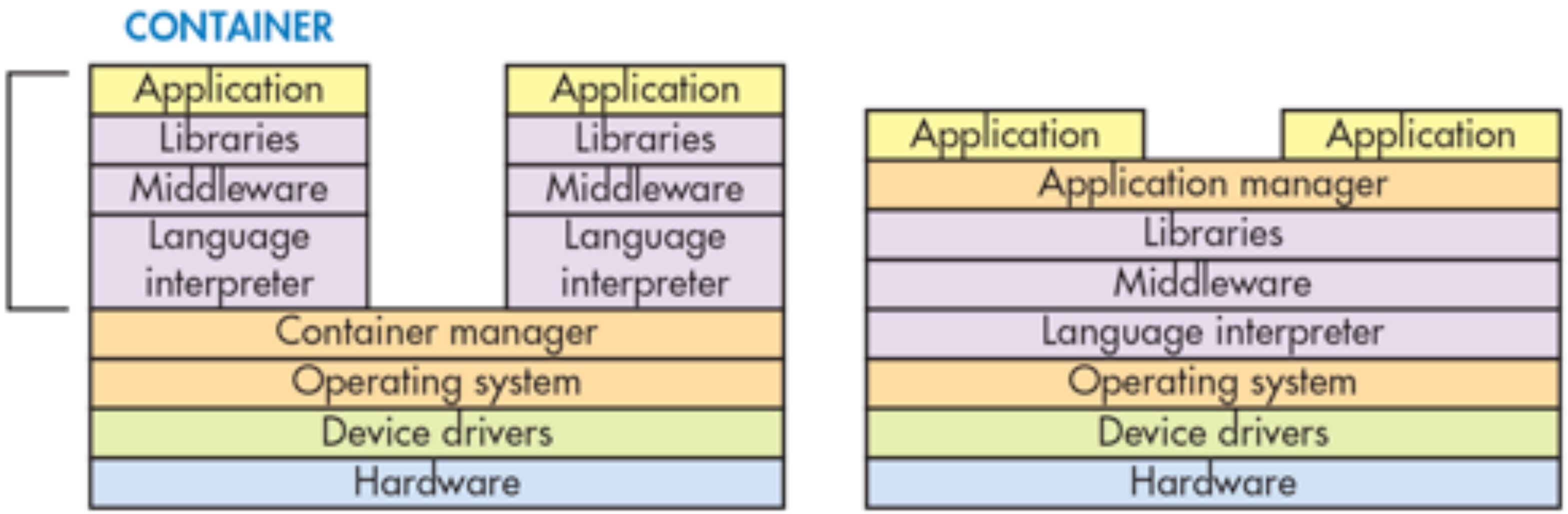

Synchronous
(push)
Amazon API
Gateway
/order
Lambda
function
Asynchronous
(event)
Amazon
SNS
Amazon
S3
reqs
Lambda
function
Stream
(Poll-based)
Amazon
DynamoDB
Amazon
Kinesis
changes
AWS Lambda
service
function

Serverless Function use cases

- Good for **infrequently-executed** code and **batch processing**.
  - No resources are wasted while the code is idle.
- Good for **bursty** workloads.
  - A serverless function can spin up in a few seconds.
    - It's just a code copy and the code startup time.
    - This is much faster than starting a new VM (a few minutes) or container.
- For example, quickly processing a sudden burst of thousands documents.
  - To do it quickly, the work must be done in parallel on lots of machines.
  - Keeping lots of VMs available (and idle) would be costly.
  - Creating new VMs would be slow (and costly because you pay during bootup).

Computing Platform comparison





VIRTUAL MACHINES

- ✓ Run multiple isolated OSes on one machine.
- ✓ Consistent environment
- ✗ High space overhead.
- ✗ Slow to start/stop.
- ✗ Horizontal scaling and resource mgmt. is manual.

CONTAINERS

- ✓ Consistent runtime environment without overhead of a full VM.
- ✓ Automatic horiz. scaling.
- ✗ Less configurable than a VM.

SERVERLESS

- ✓ Quickest startup, least storage overhead.
- ✓ Automatic horiz. scaling.
- ✗ Must work entirely in one programming language.
- ✗ Must be stateless, <15min.
- ✗ Cloud vendor lock-in.

Cloud architecture tutorials from AWS

- https://github.com/aws-samples/aws-modern-application-workshop/tree/java
- https://github.com/aws-samples/aws-serverless-airline-booking

Tips:

- **API Gateway** is an alternative way to develop APIs, using Lambdas:
  - https://us-east-2.console.aws.amazon.com/apigateway/home
- Long-running processing work can be done in ECS on Fargate:
  - https://kalinchernev.github.io/solving-aws-lambda-timeouts-fargate

Recap – Computing Platforms

- **Virtual Machines** let multiple tenants share a single physical server.
  - Apps can be distributed as a **VM disk image.**
- **Containers** create a consistent environment for your application.
  - Distribute as an **image** (like a VM disk image, but more lightweight),
  - Or as source code + a **Dockerfile** (a blueprint for the image).
- **Serverless functions** are code that is *staged* in the cloud, *ready to run:*
  - They are automatically deployed and run on *one or more* VMs **on demand**.

Pros:
- Gives more dynamic & fine-grained scalability than container/VM.
- Uses as many machines as needed, when needed. (zero to 1000s!)

Cons:
- "Cold start" delay of several seconds (to copy code & launch).
- Function runtime is often limited (eg., 15 minutes for Lambda).