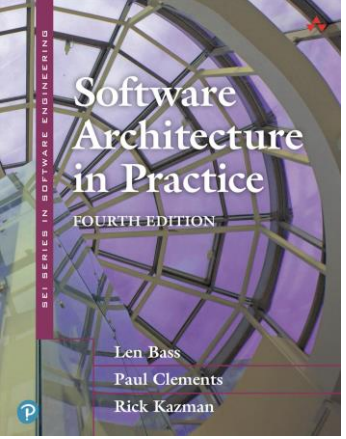
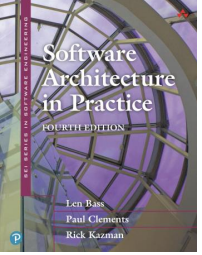


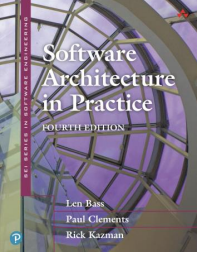
Chapter 17: The Cloud and Distributed Computing

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
—Leslie Lamport



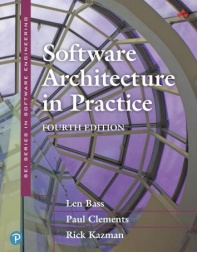
Chapter Outline

- Cloud Basics
- Failure in the Cloud
- Using Multiple Instances to Improve Performance and Availability
- Summary



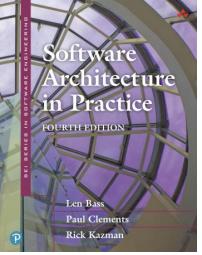
Cloud Basics

- *Public clouds* are run by cloud service providers. These organizations provide services to anyone who agrees to the terms of service and can pay. These services are typically accessible on the Internet, although you can use mechanisms such as firewalls to restrict access.
- *Private clouds* are owned and operated by an organization for its own use. An organization might choose to host a private cloud because of concerns such as control, security, and cost.
- *Hybrid cloud* approaches have some workloads run in a private cloud and others in a public cloud.



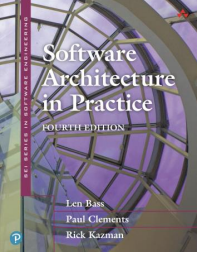
Cloud Data Center

- A typical public cloud data center has tens of thousands of physical devices—closer to 100,000 than to 50,000.
- Each rack in a data center consists of more than 25 computers (each with multiple CPUs), with the exact number depending on the power and cooling available.



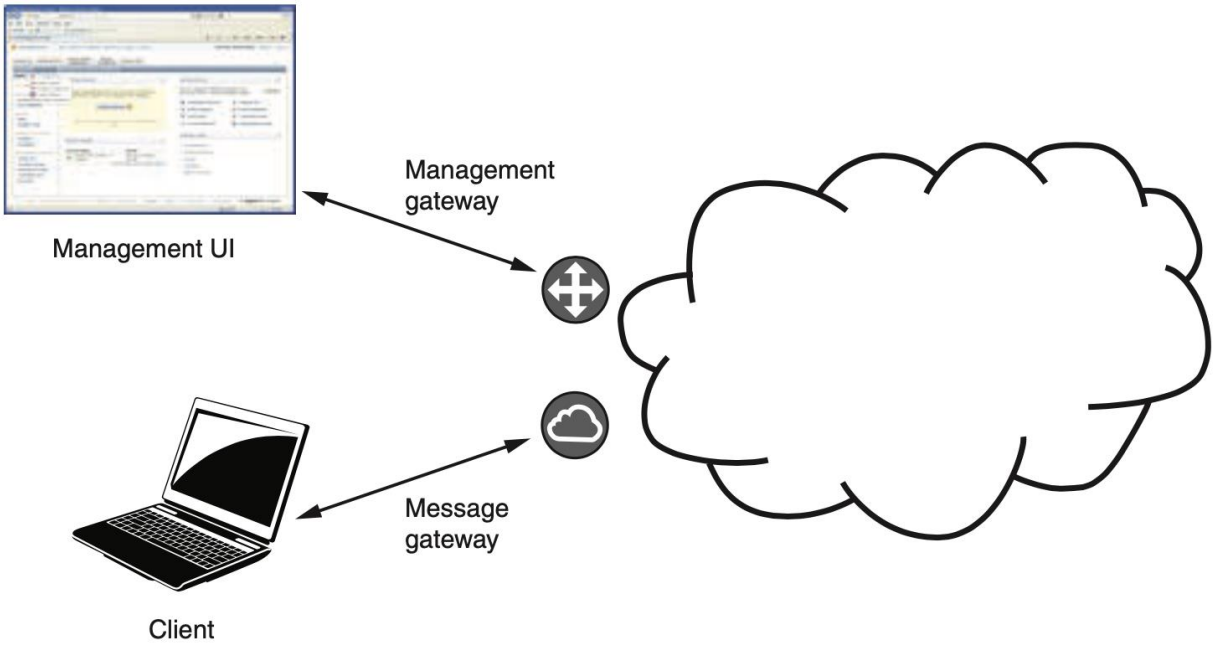
Cloud Data Center

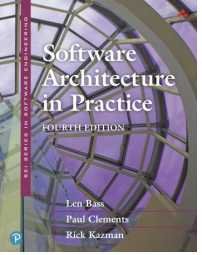
- When you access a cloud via a public cloud provider, you are actually accessing data centers scattered around the globe.
- The cloud provider organizes its data centers into *regions*.
- The data centers within a region are grouped into *availability zones*, such that the probability of all data centers failing at the same time is low.
- Choosing the cloud region that your service will run on is an important design decision.



Gateways in a Public Cloud

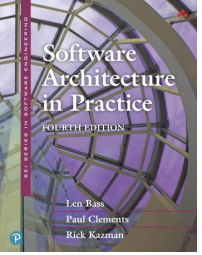
- Suppose you wish to have a VM allocated for you in the cloud.
- You send a request to a *management gateway* asking for a new VM instance.
- This request has many parameters, but three essential ones are: the cloud region where the new instance will run, the instance type (e.g., CPU and memory size), and the ID of a VM image.





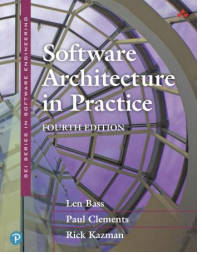
Gateways in a Public Cloud

- The management gateway is responsible for tens of thousands of physical computers, and each physical computer has a hypervisor that manages its VMs.
- The management gateway will identify a hypervisor that can manage an additional VM instance.
- The management gateway performs other functions in addition to allocating VMs, e.g. collecting billing information, and providing the capability to monitor and destroy the VM.



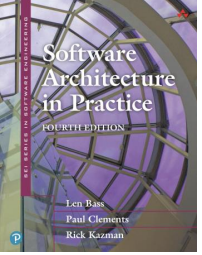
Failure in the Cloud

- When a data center contains tens of thousands of physical computers; some fail every day.
- One day it will be the computer your service runs on.
- How should you think about and manage this?



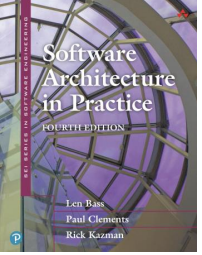
Timeouts

- Recall from Chapter 4 that timeout is a tactic for availability. In a distributed system, timeouts are used to detect failure.
- There are several consequences of using timeouts:
  - Timeouts can't distinguish between a failed computer or broken network connection and a slow reply to a message that exceeds the timeout period.
  - A timeout will not tell you *where* the failure or slowness occurs.
  - Service requests may trigger other requests; if each response in the chain is slow, the overall latency may (falsely) suggest a failure.



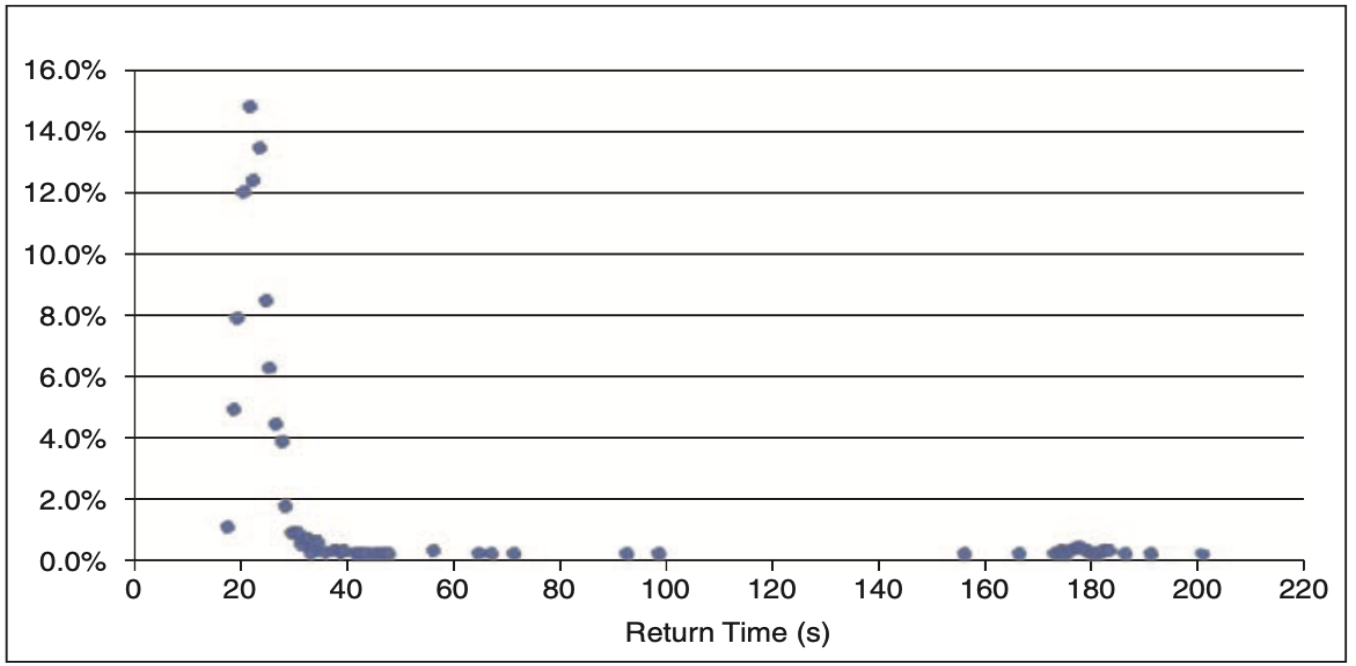
Timeouts

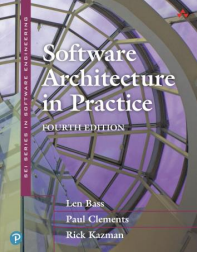
- How to handle this? Start a new VM? Switch to a different service instance?
- For systems running in a single data center, timeouts and thresholds can be set aggressively, since network delays are minimal and missed responses are likely due to software crashes or hardware failures.
- For systems operating over slower networks, more thought should be put into setting the parameters, to avoid triggering unnecessary recovery actions.



Long Tail Latency

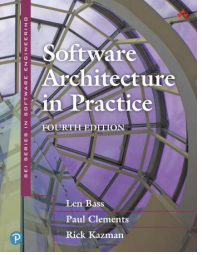
- Here is a histogram of the latency of 1,000 “launch instance” requests to Amazon Web Services (AWS).
- Some requests took a very long time to satisfy.





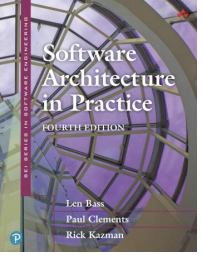
Long Tail Latency

- Long tail latencies are a result of congestion or failure somewhere in the path of the service request.
- Two techniques to handle these problems are:
  - *Hedged requests*. Make more requests than are needed and then cancel the requests (or ignore responses) after sufficient responses have been received.
  - *Alternative requests.* A variant of the hedged request technique is called alternative request. In the just-described scenario, issue 10 requests. When 8 requests have completed, issue 2 more, and when a total of 10 responses have been received, cancel the 2 requests that are still remaining.



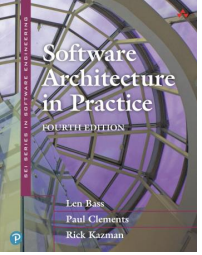
Using Multiple Instances to Improve Performance and Availability

- If a service hosted in a cloud receives more requests than it can process within the required latency, the service becomes overloaded.
- In some cases, this is due to insufficient resources and you can simply run the service in a different instance that provides more of that resource.
- This is called vertical scaling or scaling up (corresponding to the *increase resources* performance tactic from Chapter 9).



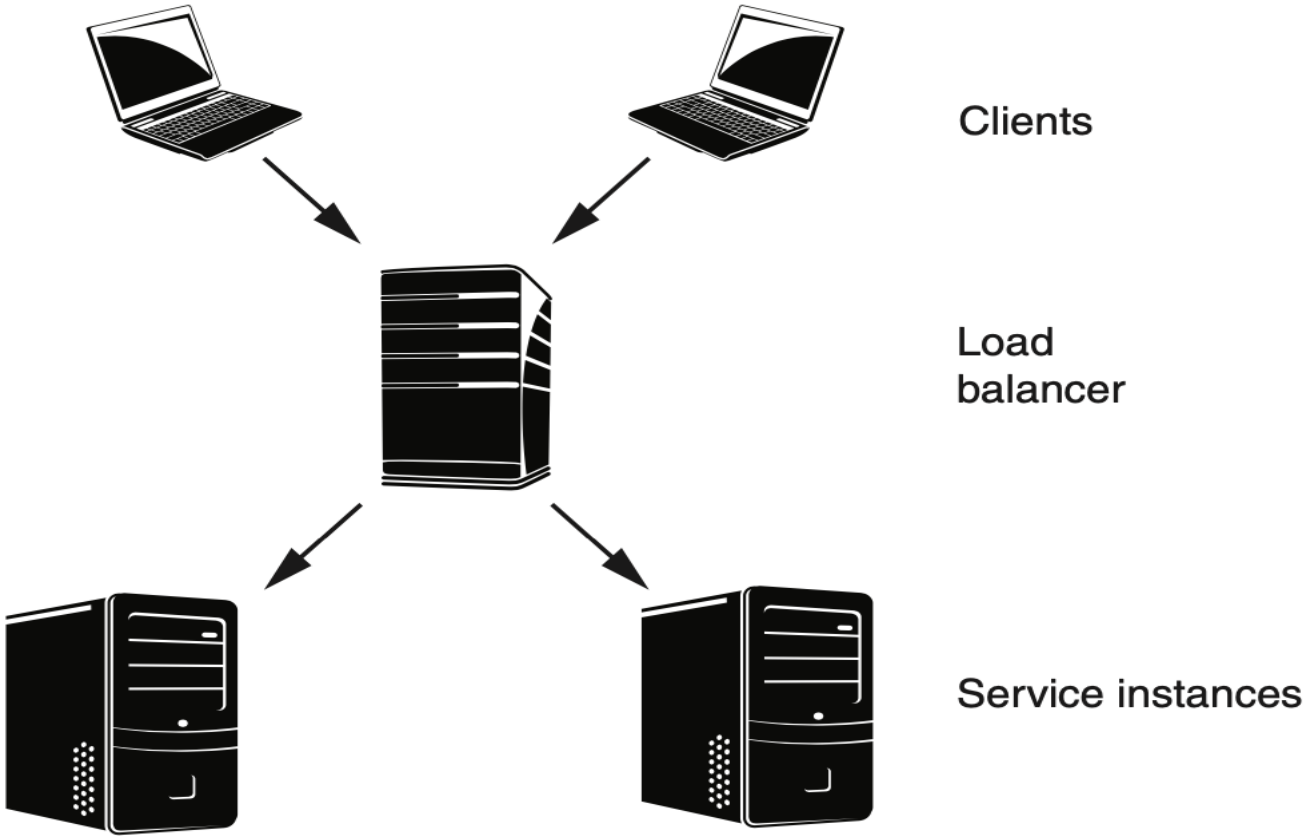
Using Multiple Instances to Improve Performance and Availability

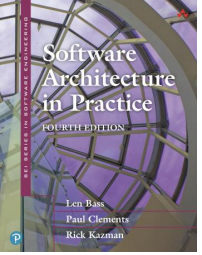
- There are limits to what can be achieved with vertical scaling—there may not be a large enough VM instance.
- In this case, horizontal scaling or scaling out is used.
- Horizontal scaling involves having multiple copies of the same service and using a load balancer to distribute requests among them (see the *maintain multiple copies of computations* tactic and the *load balancer* pattern from Chapter 9).



Load Balancers

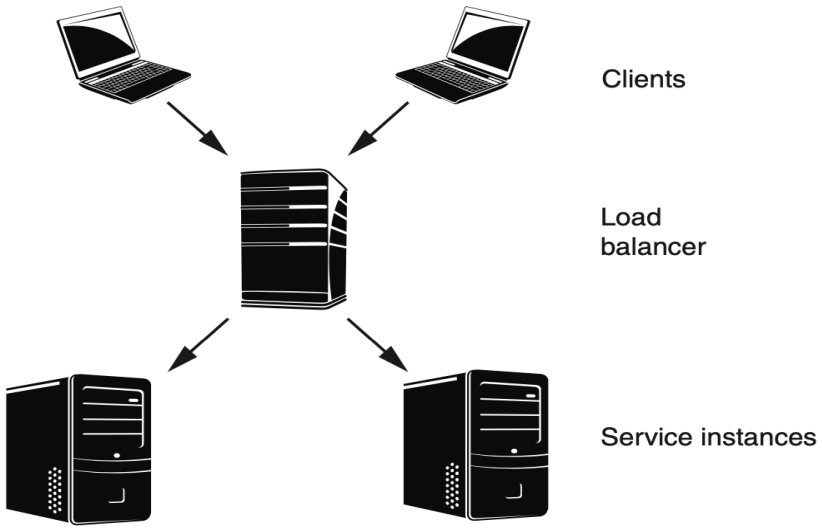
- Here is a load balancer distributing requests between two VM (service) instances.

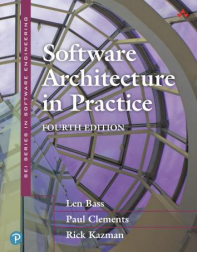




Load Balancers for Performance

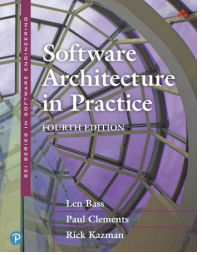
- Suppose the load balancer sends the first request to instance 1, the second to instance 2, the third to instance 1, etc.
- This sends half of the requests to each instance, *balancing* the *load* between them.





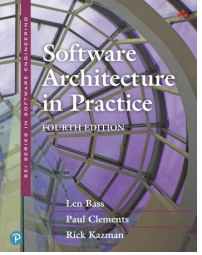
Load Balancers for Performance

- The algorithm we just sketched is called “round-robin.” This algorithm balances the load uniformly only if every request consumes roughly the same resources. Other algorithms for cases where resource consumption to process requests varies.
- From a client’s perspective, the service’s IP address is actually the address of the load balancer. The client does not need to know how many instances of the service exist or the IP address of any of those instances.
- Multiple clients may coexist. The load balancer distributes the messages as they arrive.
- Load balancers may get overloaded. The solution is to balance the load among multiple load balancers, sometimes referred to as global load balancing.



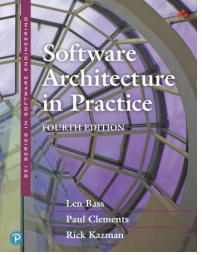
Load Balancers for Availability

- Load balancers may have no information about whether a message was processed, or how long the processing took.
- Without additional mechanisms, the load balancer would not know whether any instance or all instances had failed.



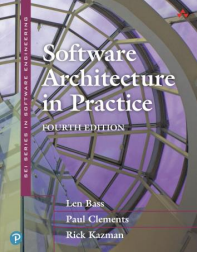
Load Balancers for Availability

- Health checks allow the load balancer to determine whether an instance is performing properly, using “fault detection” availability tactics (see Chapter 4).
- The load balancer will periodically check the health of the instances assigned to it. Failed instances are marked as unhealthy and no further messages are sent to them.
- Health checks can consist of pings from the load balancer to the instance, opening a TCP connection to the instance or even sending a message for processing.



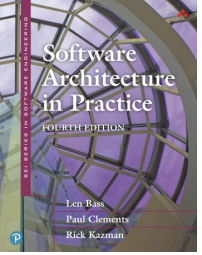
Load Balancers for Availability

- A load balancer with health checking improves availability by hiding the failure of an instance from clients.
- The pool of service instances can be sized to accommodate some failures while still maintaining the desired latency.
- However, a service instance might start processing a client request but never return a response. Clients must be designed to resend failed requests.
- Services must correspondingly be designed such that multiple identical requests can be accommodated.



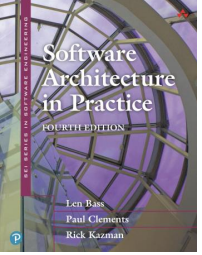
State Management

- Management of state is important when a service can process more than one client request at the same time.
- Where should state be stored?
  - history is maintained in each instance, in which case the services are “stateful.”
  - history is maintained in each client, in which case the services are “stateless.”
  - history persists outside the services and clients, in a database, in which case the services are “stateless.”



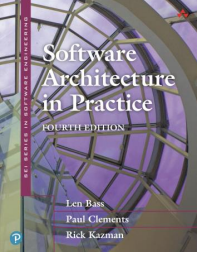
Time Coordination

- Determining exactly what time it is can be challenging. Having two or more devices agree on what time it is can be even more challenging.
- You should assume some level of error exists between the clock readings on two different devices.
- Most distributed systems are designed so that time synchronization is not required to function correctly—it is more important to know the *order* of events.
- For an architect, successful time coordination involves knowing whether you need to rely on actual clock times, or whether ensuring correct sequencing suffices.



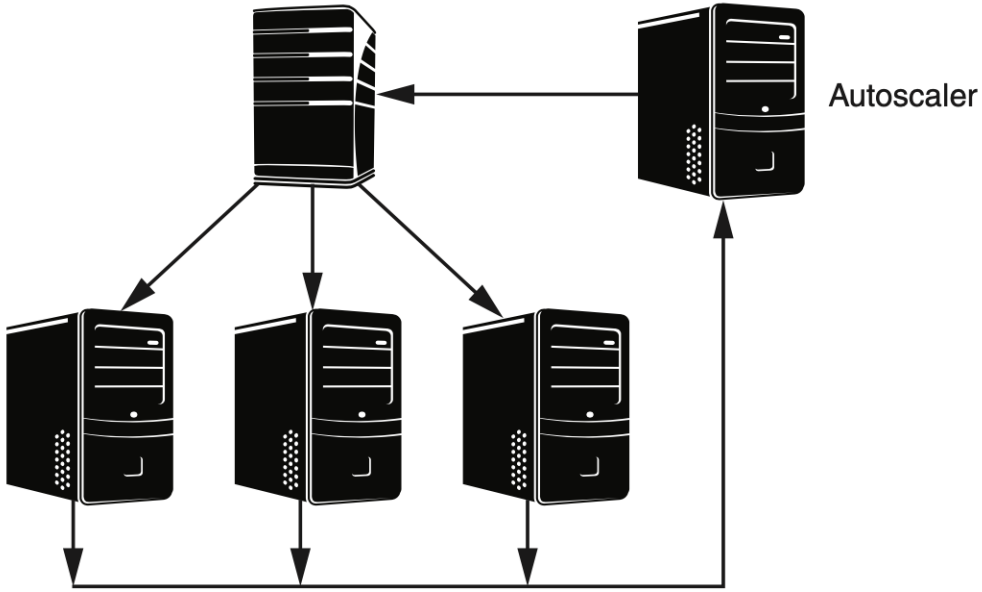
Data Coordination

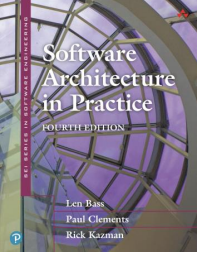
- Data coordination is complex in a distributed system for two reasons:
  - the traditional two-phase commit protocol requires multiple messages to be transmitted. This adds latency and complexity (e.g. dealing with potential for lost messages).
  - instance 1 may fail after it has acquired the lock, preventing instance 2 from proceeding.
- Existing solution packages, such as Apache Zookeeper, Consul, and etcd, solve many of these problems.



Autoscaling

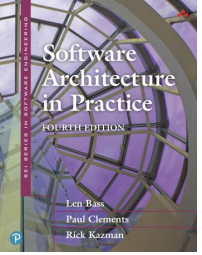
- Autoscaling services automatically create new instances when needed and release them when not needed.
- Autoscaling works differently for VMs and containers.





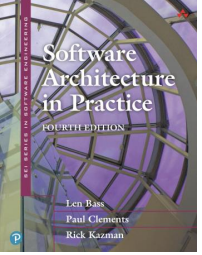
Autoscaling VMs

- Because the clients do not know how many instances exist or which instance is serving their requests, autoscaling activities are invisible to them.
- As an architect of a cloud-based service, you can set up a collection of rules for the autoscaler that govern its behavior, includeing:
  - The VM image to be launched when a new instance is created, and any instance configuration parameters
  - The CPU utilization threshold above which a new instance is launched
  - The CPU utilization threshold below which an existing instance is shut down
  - The network I/O bandwidth thresholds for creating and deleting instances
  - The minimum and maximum number of instances you want in this group



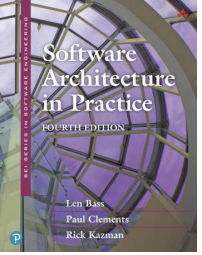
Autoscaling Containers

- Because containers are executing on runtime engines that are hosted on VMs, scaling containers involves two types of decisions.
- Scaling containers means making a two-level decision:
  1. decide that an additional container (or Pod) is required for the current workload.
  2. decide whether the new container (or Pod) can be allocated on an existing runtime engine instance or whether a new instance must be allocated.
- The software that controls the scaling of containers is independent of the software that controls the scaling of VMs.



Summary

- The cloud is composed of distributed data centers, managed through a management gateway. It is responsible for allocating, deallocating, and monitoring VMs, as well as measuring resource usage and computing billing.
- You should assume that at some point, the VMs on which your service is executing will fail.
- You should also assume that your requests for other services will exhibit a long tail distribution.
- Thus you must be concerned about the availability of your service.



Summary

- Because single instances of your service may not be able to satisfy all requests, you may decide to run multiple VMs or containers.
- These instances sit behind a load balancer.
- The existence of multiple instances of your service and multiple clients has a significant impact on how you handle state.
- The cloud infrastructure can automatically scale your service by creating new instances when demand grows and removing instances when demand shrinks.