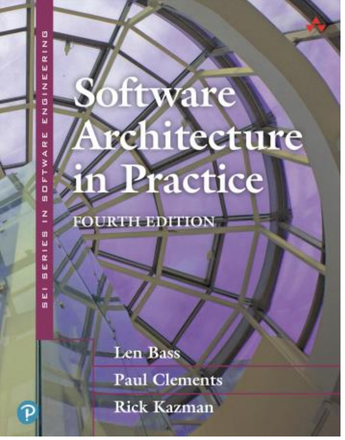
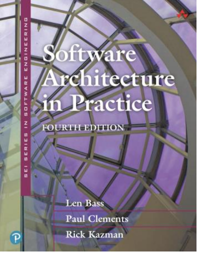

Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

Chapter 9: Performance

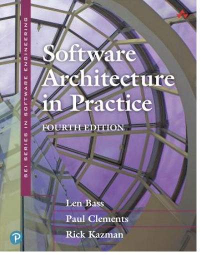
An ounce of performance is worth pounds of promises.

—Mae West



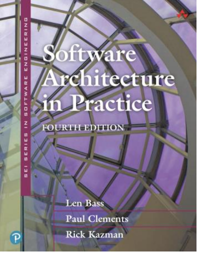
Chapter Outline

- What is Performance?
- Performance General Scenario
- Tactics for Performance
- Tactics-Based Questionnaire for Performance
- Patterns for Performance
- Summary



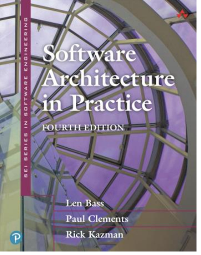
What is Performance?

- Performance is about time and the software system’s ability to meet timing requirements.
- When events occur – interrupts, messages, requests from users or other systems, or clock events marking the passage of time – the system, or some element of the system, must respond to them in time.
- Characterizing the events that can occur (and when they can occur) and the system or element’s time-based response to those events is the essence is discussing performance.



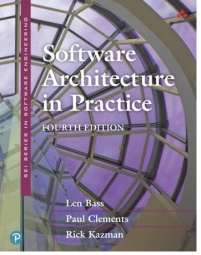
What is Performance?

- All systems have performance requirements, even if they are not expressed.
- Performance is often linked to scalability—that is, increasing your system’s capacity for work, while still performing well.
- Often, performance improvement happens after you have constructed a version of your system and found its performance to be inadequate. You can anticipate this by architecting your system with performance in mind.



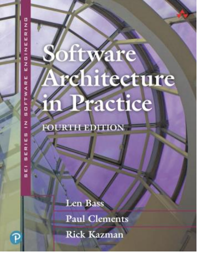
Performance General Scenario

Portion of Scenario	Description	Possible Values
Source	The stimulus can come from a user (or multiple users), from an external system, or from some portion of the system under consideration.	External: ▪ User request ▪ Request from external system ▪ Data arriving from a sensor or other system Internal: ▪ One component may make a request of another component. ▪ A timer may generate a notification.
Stimulus	The stimulus is the arrival of an event. The event can be a request for service or a notification of some state of either the system under consideration or an external system.	Arrival of a periodic, sporadic, or stochastic event: ▪ A periodic event arrives at a predictable interval. ▪ A stochastic event arrives according to some probability distribution. ▪ A sporadic event arrives according to a pattern that is neither periodic nor stochastic.
Artifact	The artifact stimulated may be the whole system or just a portion of the system. For example, a power-on event may stimulate the whole system. A user request may arrive at (stimulate) the user interface.	▪ Whole system ▪ Component within the system



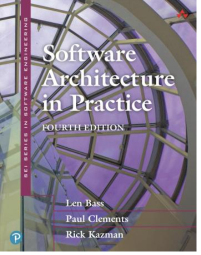
Performance General Scenario

Environment	The state of the system or component when the stimulus arrives. Unusual modes—error mode, overloaded mode—will affect the response. For example, three unsuccessful login attempts are allowed before a device is locked out.	Runtime. The system or component can be operating in: ▪ Normal mode ▪ Emergency mode ▪ Error correction mode ▪ Peak load ▪ Overload mode ▪ Degraded operation mode ▪ Some other defined mode of the system
Response	The system will process the stimulus. Processing the stimulus will take time. This time may be required for computation, or it may be required because processing is blocked by contention for shared resources. Requests can fail to be satisfied because the system is overloaded or because of a failure somewhere in the processing chain.	▪ System returns a response ▪ System returns an error ▪ System generates no response ▪ System ignores the request if overloaded ▪ System changes the mode or level of service ▪ System services a higher-priority event ▪ System consumes resources
Response measure	Timing measures can include latency or throughput. Systems with timing deadlines can also measure jitter of response and ability to meet the deadlines. Measuring how many of the requests go unsatisfied is also a type of measure, as is how much of a computing resource (e.g., a CPU, memory, thread pool, buffer) is utilized.	▪ The (maximum, minimum, mean, median) time the response takes (latency) ▪ The number or percentage of satisfied requests over some time interval (throughput) or set of events received ▪ The number or percentage of requests that go unsatisfied ▪ The variation in response time (jitter) ▪ Usage level of a computing resource

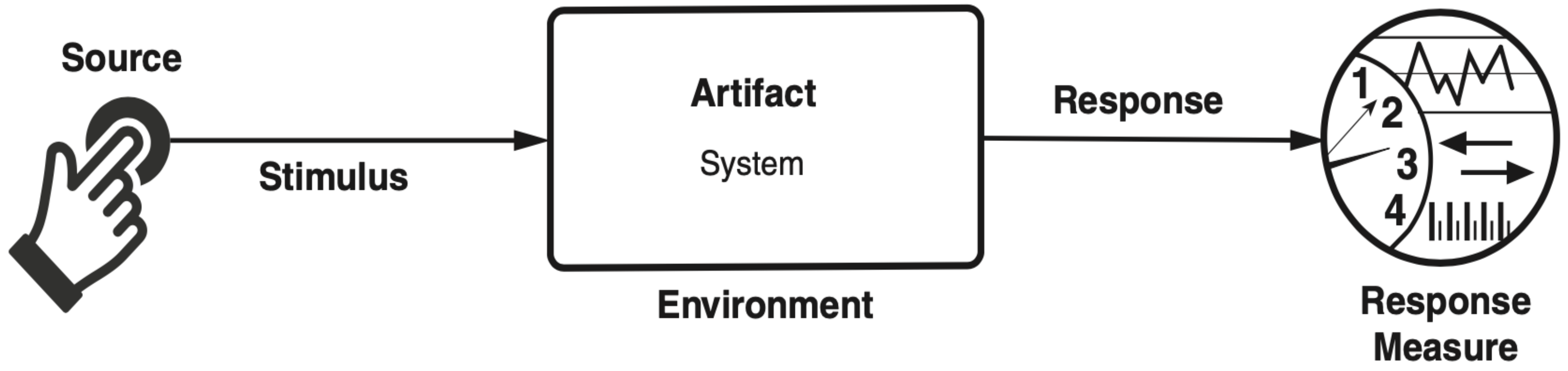


Sample Concrete Performance Scenario

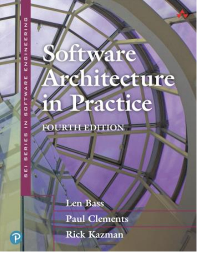
- *Five hundred users initiate 2,000 requests in a 30-second interval, under normal operations. The system processes all of the requests with an average latency of two seconds.*



Sample Concrete Performance Scenario

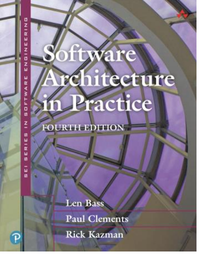


500 users	Initiate 2,000 requests in a 30-second interval	Normal operations	Processes all requests	Average latency of 2 seconds

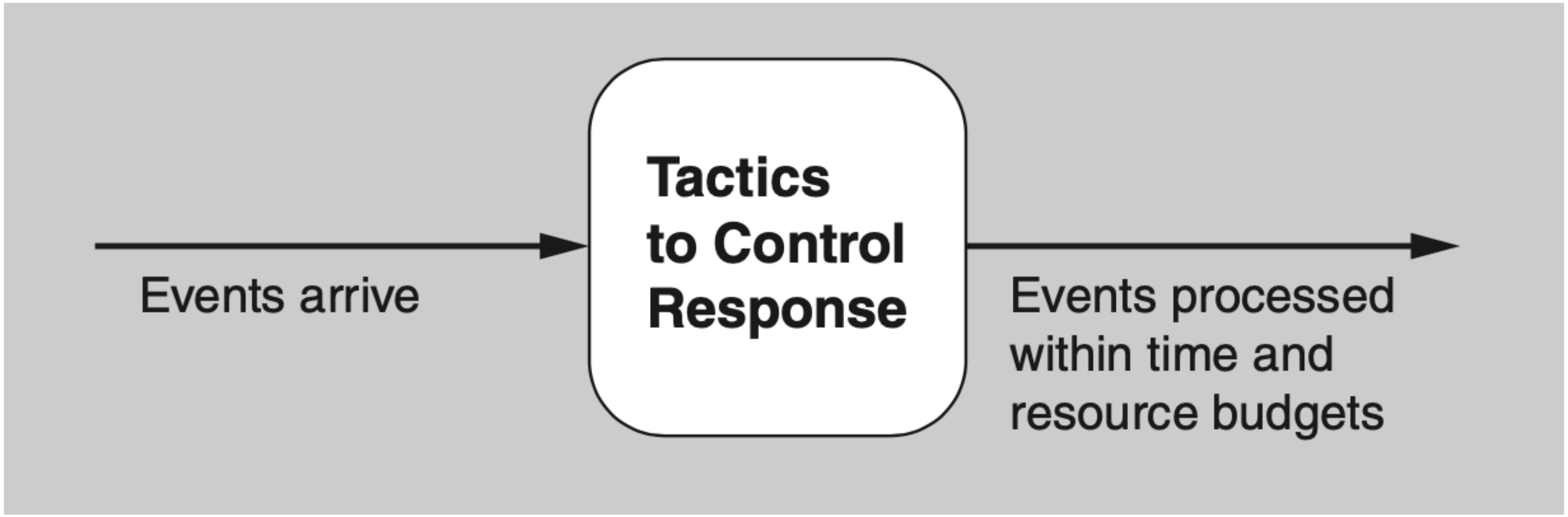


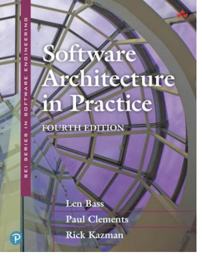
Goal of Performance Tactics

- The goal of performance tactics is to generate a response to events arriving at the system under some time-based or resource-based constraint.
- The event can be a single event or a stream, and is the trigger to perform computation.
- Performance tactics control the time or resources used to generate a response.

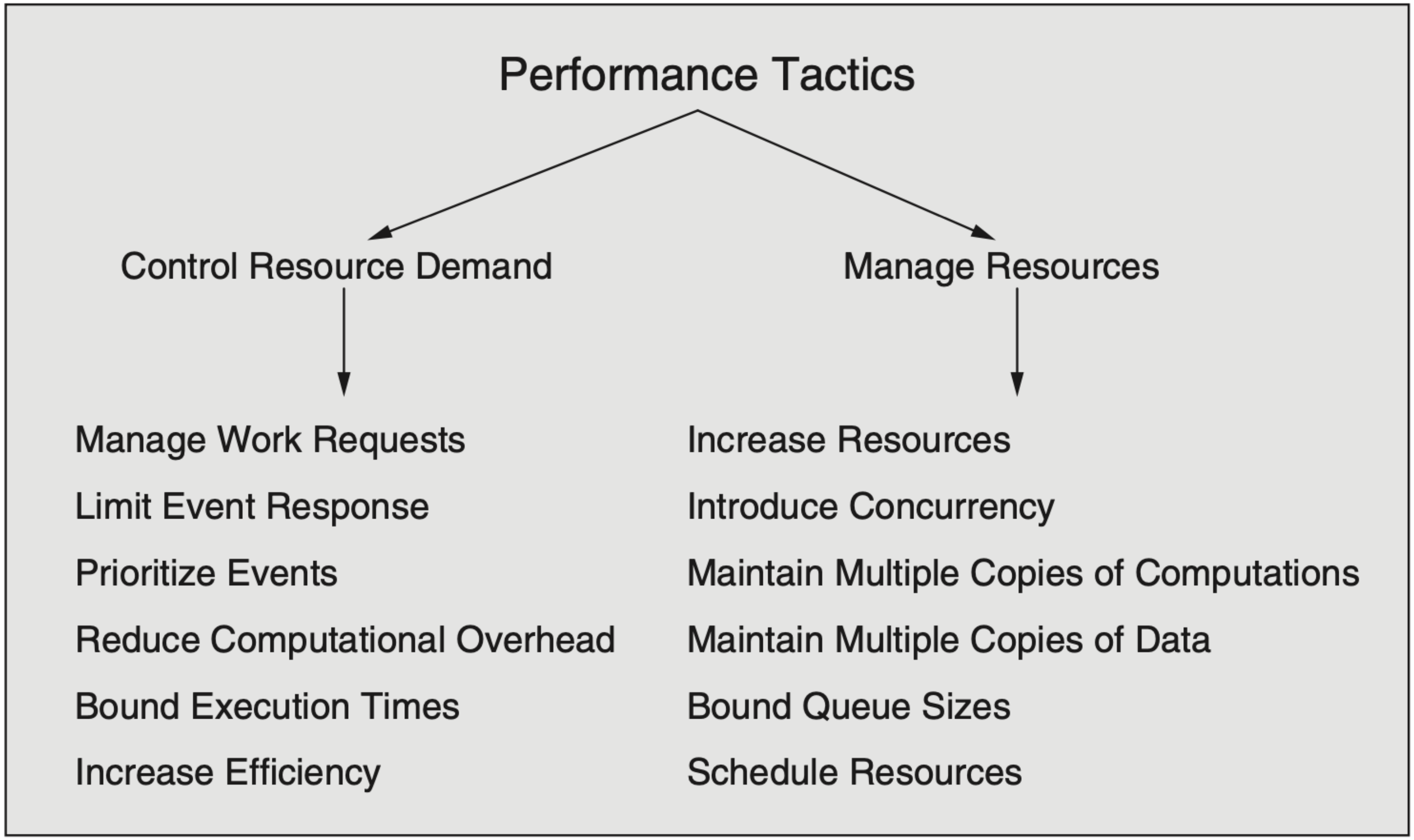


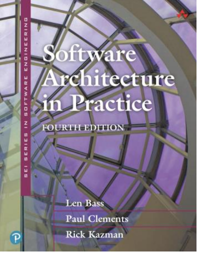
Goal of Performance Tactics


Events arrive
Tactics to Control Response
Events processed within time and resource budgets



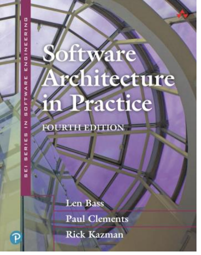
Performance Tactics





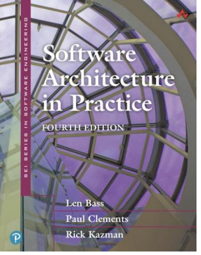
Control Resource Demand

- *Manage work requests.* One way to reduce work is to reduce the number of requests coming into the system to do work. Ways to do that include:
  - Manage event arrival
  - Manage sampling rate
- *Limit event response*. When events arrive too rapidly to be processed, they must be queued until they can be processed, or they are discarded. You may choose to process events only up to a set maximum rate, thereby ensuring predictable processing for other events. This tactic could be triggered by a queue size or processor utilization exceeding a warning level. Or it could be triggered by an event rate that violates an SLA.
- *Prioritize events*. If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them. If insufficient resources are available to service them when they arise, low-priority events might be ignored.



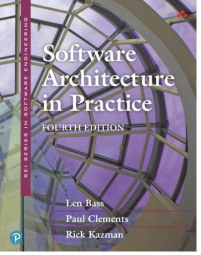
Control Resource Demand

- *Reduce computational overhead.* For events that do make it into the system, the following approaches can be implemented to reduce the amount of work involved in handling each event:
  - Reduce indirection
  - Co-locate communicating resources
  - Periodic cleaning
- *Bound execution times*. You can place a limit on how much execution time is used to respond to an event.
- *Increase efficiency of resource usage.* Improving the efficiency of algorithms used in critical areas can decrease latency and improve throughput and resource consumption.



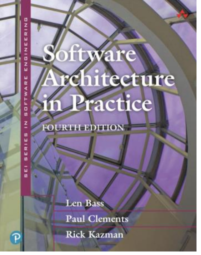
Manage Resources

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential to improve performance. Cost is usually a consideration in the choice of resources, but increasing the resources is, in many cases, the cheapest way to get immediate improvement.
- *Introduce concurrency*. If requests can be processed in parallel, blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.
- *Maintain multiple copies of computations*. This tactic reduces the contention that would occur if all requests for service were allocated to a single instance.

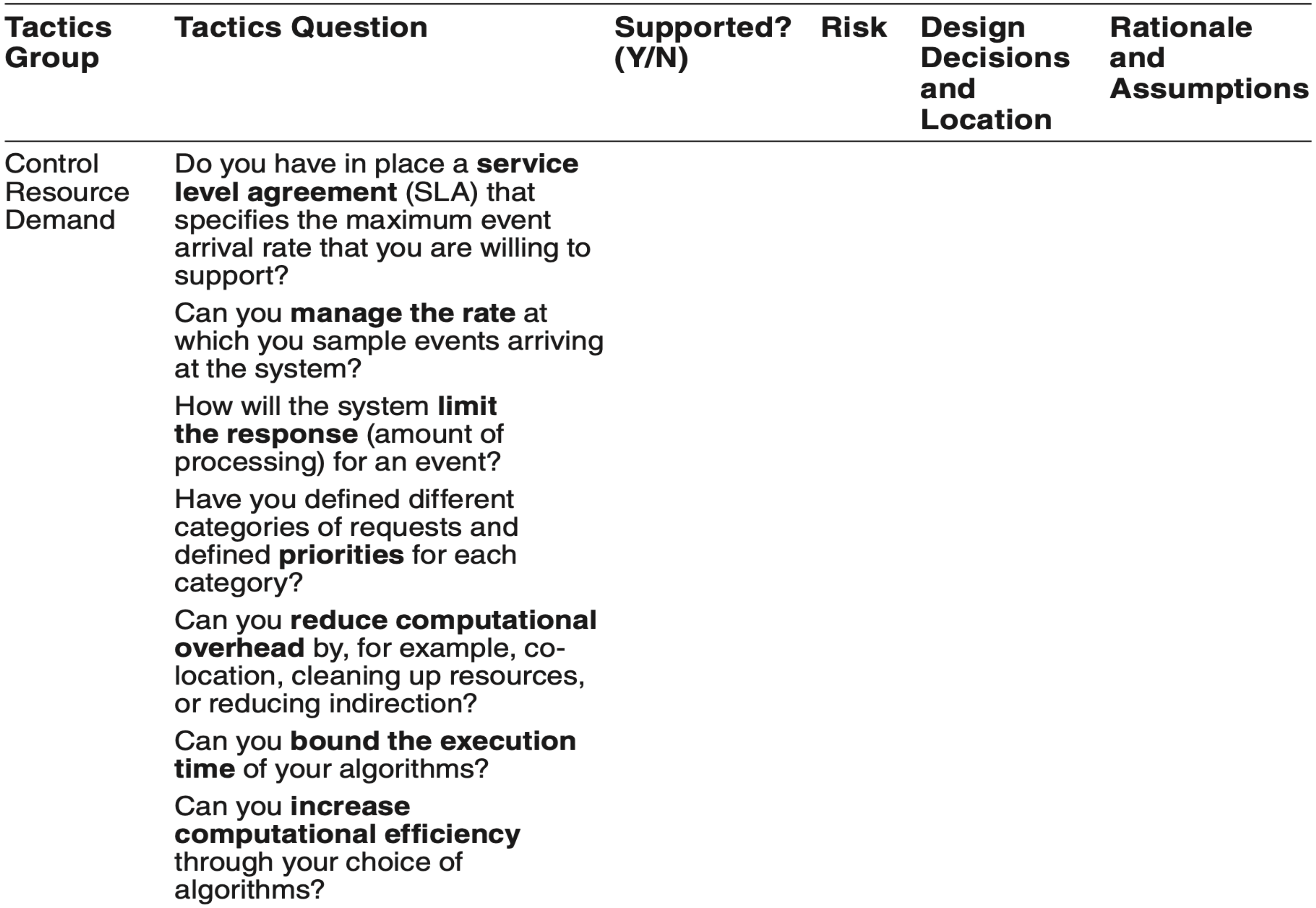


Manage Resources

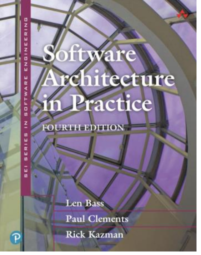
- *Maintain multiple copies of data.* Two common examples of maintaining multiple copies of data are data replication and caching. *Data replication* involves keeping separate copies of the data to reduce contention. *Caching* also involves keeping copies of data (with one set of data possibly being a subset of the other), but on storage with different access speeds.
- *Bound queue sizes*. This tactic controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to establish a policy for what happens when the queues overflow and decide if not responding to lost events is acceptable.
- *Schedule resources*. Whenever contention for a resource occurs, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your concern as an architect is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it.



Tactics-Based Questionnaire for Performance

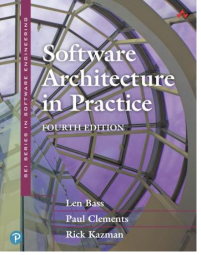


Tactics Group	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Control Resource Demand	Do you have in place a **service level agreement** (SLA) that specifies the maximum event arrival rate that you are willing to support?				
	Can you **manage the rate** at which you sample events arriving at the system?				
	How will the system **limit the response** (amount of processing) for an event?				
	Have you defined different categories of requests and defined **priorities** for each category?				
	Can you **reduce computational overhead** by, for example, co-location, cleaning up resources, or reducing indirection?				
	Can you **bound the execution time** of your algorithms?				
	Can you **increase computational efficiency** through your choice of algorithms?				



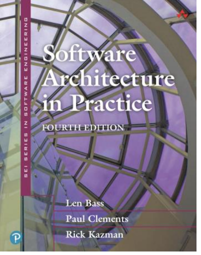
Tactics-Based Questionnaire for Performance

Manage Resources	Can you **allocate more resources** to the system or its components?
	Are you employing **concurrency**? If requests can be processed in parallel, the blocked time can be reduced.
	Can **computations be replicated** on different processors?
	Can **data** be **cached** (to maintain a local copy that can be quickly accessed) or replicated (to reduce contention)?
	Can **queue sizes be bounded** to place an upper bound on the resources needed to process stimuli?
	Have you ensured that the **scheduling strategies** you are using are appropriate for your performance concerns?



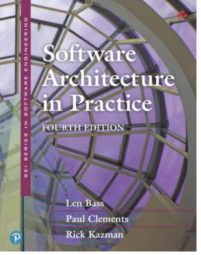
Service Mesh Pattern for Performance

- The service mesh pattern is used in microservice architectures. The main feature is a *sidecar*—a proxy that accompanies each microservice which provides capabilities such as interservice communications, monitoring, and security.
- A sidecar executes alongside each microservice and handles all interservice communication and coordination. They are deployed together, which cuts down on the latency due to networking, thereby boosting performance.



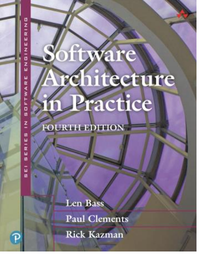
Service Mesh Pattern Benefits

- Benefits:
  - Software to manage cross-cutting concerns can be purchased off the shelf or implemented by a specialist team that does nothing else, allowing developers of the business logic to focus on just that.
  - A service mesh enforces the deployment of utility functions onto the same processor as the services that use those utility functions. This cuts down on communication.
  - The service mesh can be configured to make communication dependent on context, thus simplifying functions such as the canary and A/B testing described in Chapter 3.



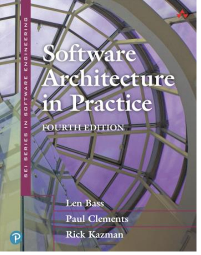
Service Mesh Pattern Tradeoffs

- Tradeoffs:
  - The sidecars introduce more executing processes, and each of these will consume some processing power, adding to the system’s overhead.
  - A sidecar typically includes multiple functions, and not all of these will be needed in every service or every invocation of a service.



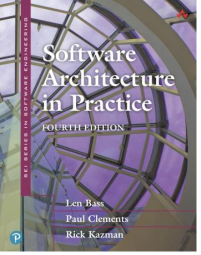
Load Balancer Pattern for Performance

- A load balancer is a kind of intermediary that handles messages originating from some set of clients and determines which instance of a service should respond to those messages.
- The load balancer serves as a single point of contact for incoming messages and farms out requests to a pool of providers that can respond to the request.
- In this way, the load can be balanced across the pool of providers.



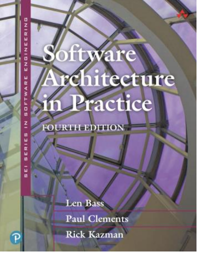
Load Balancer Pattern Benefits

- Any failure of a server is invisible to clients (assuming there are still some remaining processing resources).
- By sharing the load among several providers, latency can be kept lower and more predictable for clients.
- It is relatively simple to add more resources to the pool available to the load balancer, and no client needs to be aware of this.



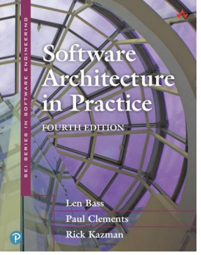
Load Balancer Pattern Tradeoffs

- The load balancing algorithm must be very fast; otherwise, it may itself contribute to performance problems.
- The load balancer is a potential bottleneck or single point of failure, so it is itself often replicated (and even load balanced).



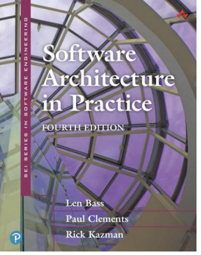
Throttling Pattern for Performance

- The throttling pattern is a packaging of the manage work requests tactic.
- It is used to limit access to some important resource or service.
- In this pattern, there is typically an intermediary—a throttler—that monitors (requests to) the service and determines whether an incoming request can be serviced.



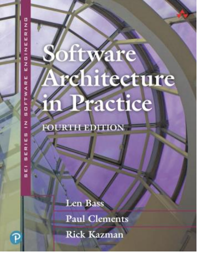
Throttling Pattern Benefits

- By throttling incoming requests, you can gracefully handle variations in demand.
- In doing so, services never become overloaded; they can be kept in a performance “sweet spot” where they handle requests efficiently.


Software Architecture in Practice
FOURTH EDITION
SEI SERIES IN SOFTWARE ENGINEERING
Len Bass
Paul Clements
Rick Kazman

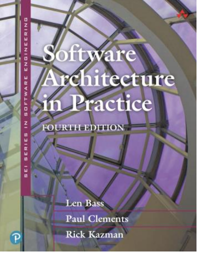
Throttling Pattern Tradeoffs

- The throttling logic must be very fast; otherwise, it may itself contribute to performance problems.
- If client demand regularly exceeds capacity, buffers will need to be very large, or there is a risk of losing requests.
- This pattern can be difficult to add to an existing system where clients and servers are tightly coupled.



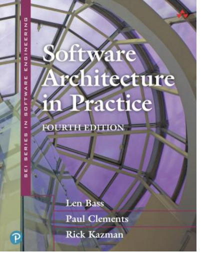
Map-Reduce Pattern for Performance

- The map-reduce pattern efficiently performs a distributed and parallel sort of a large data set and provides a simple means for the programmer to specify the analysis to be done.
- The map-reduce pattern has three parts:
  - a specialized infrastructure that takes care of allocating software to hardware nodes in a massively parallel computing environment and handles sorting the data.
  - a *map* function takes as input a key and a data set. It uses the key to hash the data into a set of buckets.
  - a *reduce* function. The number of reduce instances corresponds to the number of buckets output by the map function. The reduce phase does programmer-specified analysis and then emits the results of that analysis.



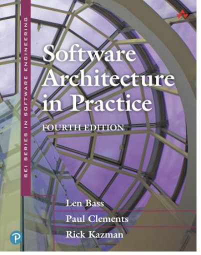
Map-Reduce Pattern Benefits

- Extremely large, unsorted data sets can be efficiently analyzed through the exploitation of parallelism.
- A failure of any instance has only a small impact on the processing, since map-reduce typically breaks large input datasets into many smaller ones for processing, allocating each to its own instance.



Map-Reduce Pattern Tradeoffs

- If you do not have large data sets, the overhead incurred by the map-reduce pattern is not justified.
- If you cannot divide your data set into similarly sized subsets, the advantages of parallelism are lost.
- Operations that require multiple reduces are complex to orchestrate.



Summary

- Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior.
- Performance can be measured in terms of throughput and latency for both interactive and embedded real time systems.
- Performance can be improved by reducing demand or by managing resources more appropriately.