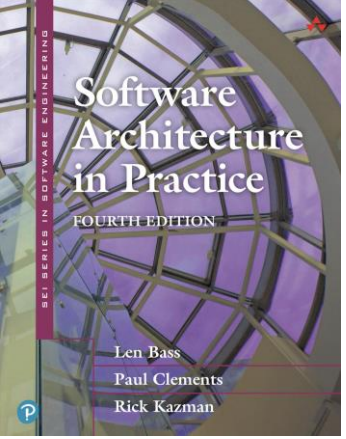
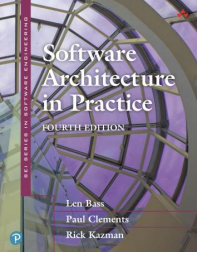


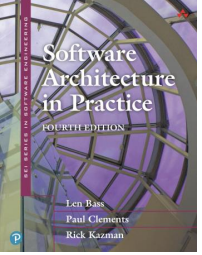
Chapter 5: Deployability

From the day we arrive on the planet
And blinking, step into the sun
There's more to be seen than can ever be seen
More to do than can ever be done
—The Lion King



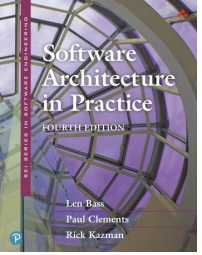
Chapter Outline

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



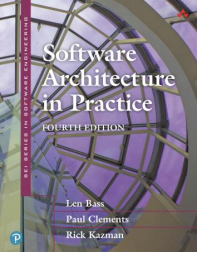
What is Deployability?

- *Deployability* refers to a property of software indicating that it may be deployed—allocated to an environment for execution—within a predictable and acceptable amount of time and effort.
- Deployment is a process that starts with coding and ends with real users interacting with the system in a production environment.
- If this process is fully automated—that is, if there is no human intervention—then it is called *continuous deployment*.



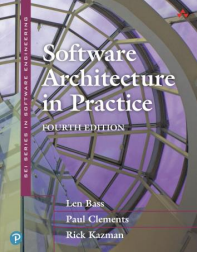
The Deployment Pipeline

- We introduce the concept of a *deployment pipeline*: the sequence of tools and activities that begin when you check your code into a version control system and end when your application has been deployed for users to send it requests.



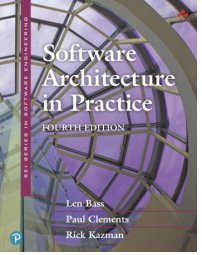
The Deployment Pipeline

- The major environments in a deployment pipeline are as follows:
  - Code is developed in a *development environment* where it is subject to standalone unit tests. When the code is committed it triggers the integration environment.
  - An *integration environment* builds an executable version of a service.
  - A *staging environment* tests for various qualities of the total system.
  - Once in the *production environment*, the service is monitored.



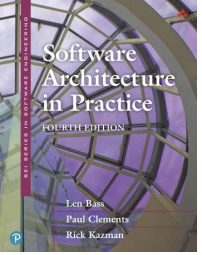
The Deployment Pipeline

- Three important ways to measure the quality of the pipeline are as follows:
  - *Cycle time* is the pace of progress through the pipeline.
  - *Traceability* is the ability to recover all of the artifacts that led to an element having a problem.
  - *Repeatability* is getting the same result when you perform the same action with the same artifacts.



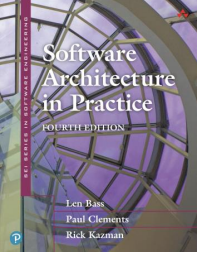
Deployability Characteristics

- Architects are concerned with the degree to which the architecture supports deployments that are:
  - *Granular:* with options for different granularities of deployment, not all-or-nothing.
  - *Controllable:* to deploy, monitor operations, and roll back unsuccessful deployments.
  - *Efficient: to* support rapid deployment (and, if needed, rollback).



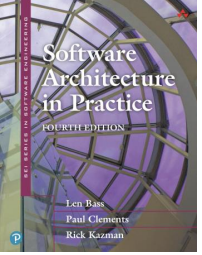
Deployability General Scenario

Portion of Scenario	Description	Possible Values
Source	The trigger for the deployment	End user, developer, system administrator, operations personnel, component marketplace, product owner.
Stimulus	What causes the trigger	A new element is available to be deployed. This is typically a request to replace a software element with a new version (e.g., fix a defect, apply a security patch, upgrade to the latest release of a component or framework, upgrade to the latest version of an internally produced element). New element is approved for incorporation. An existing element/set of elements needs to be rolled back.
Artifacts	What is to be changed	Specific components or modules, the system's platform, its user interface, its environment, or another system with which it interoperates. Thus the artifact might be a single software element, multiple software elements, or the entire system.



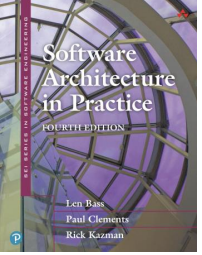
Deployability General Scenario

Environment	Staging, production (or a specific subset of either)	Full deployment. Subset deployment to a specified portion of users, VMs, containers, servers, platforms.
Response	What should happen	Incorporate the new components. Deploy the new components. Monitor the new components. Roll back a previous deployment.
Response measure	A measure of cost, time, or process effectiveness for a deployment, or for a series of deployments over time	Cost in terms of: ▪ Number, size, and complexity of affected artifacts ▪ Average/worst-case effort ▪ Elapsed clock or calendar time ▪ Money (direct outlay or opportunity cost) ▪ New defects introduced Extent to which this deployment/rollback affects other functions or quality attributes. Number of failed deployments. Repeatability of the process. Traceability of the process. Cycle time of the process.

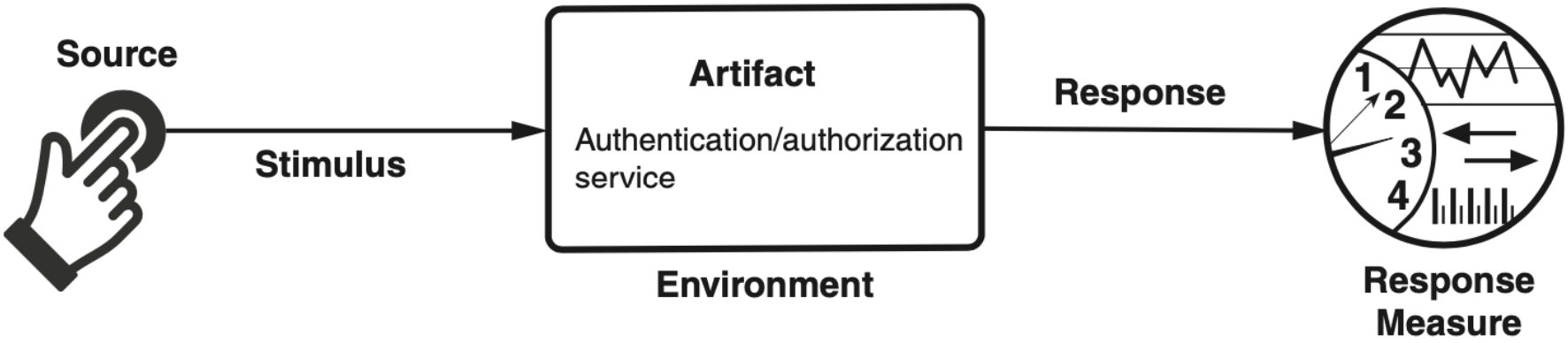


Sample Concrete Deployability Scenario

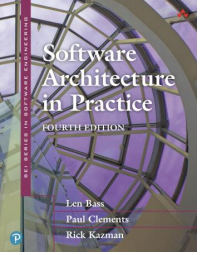
- *A new release of an authentication/ authorization service (which our product uses) is made available in the component marketplace and the product owner decides to incorporate this version into the release. The new service is tested and deployed to the production environment within 40 hours of elapsed time and no more than 120 person-hours of effort. The deployment introduces no defects and no SLA is violated.*



Sample Concrete Deployability Scenario

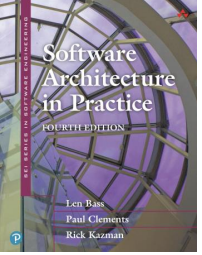


Source	Stimulus	Artifact	Environment	Response	Response Measure
Component marketplace	New release of the authentication/authorization service is made available and the product owner decides to incorporate it	Authentication/authorization service	Production	The new service is tested in-house and deployed to production servers	Within 40 hours and no more than 120 person-hours of effort No defects introduced; no SLA violated

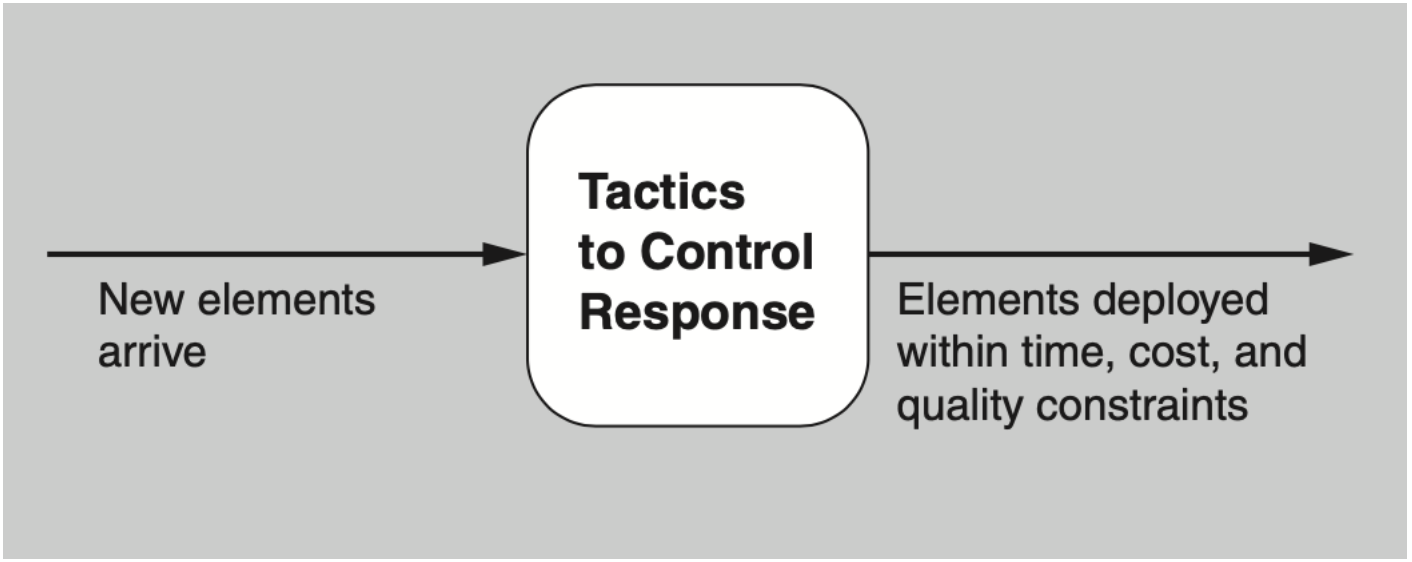


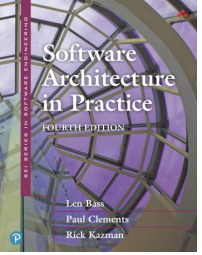
Goal of Deployability Tactics

- A deployment is catalyzed by the release of a new software or hardware element.
- The deployment is successful if these new elements are deployed within acceptable time, cost, and quality constraints.

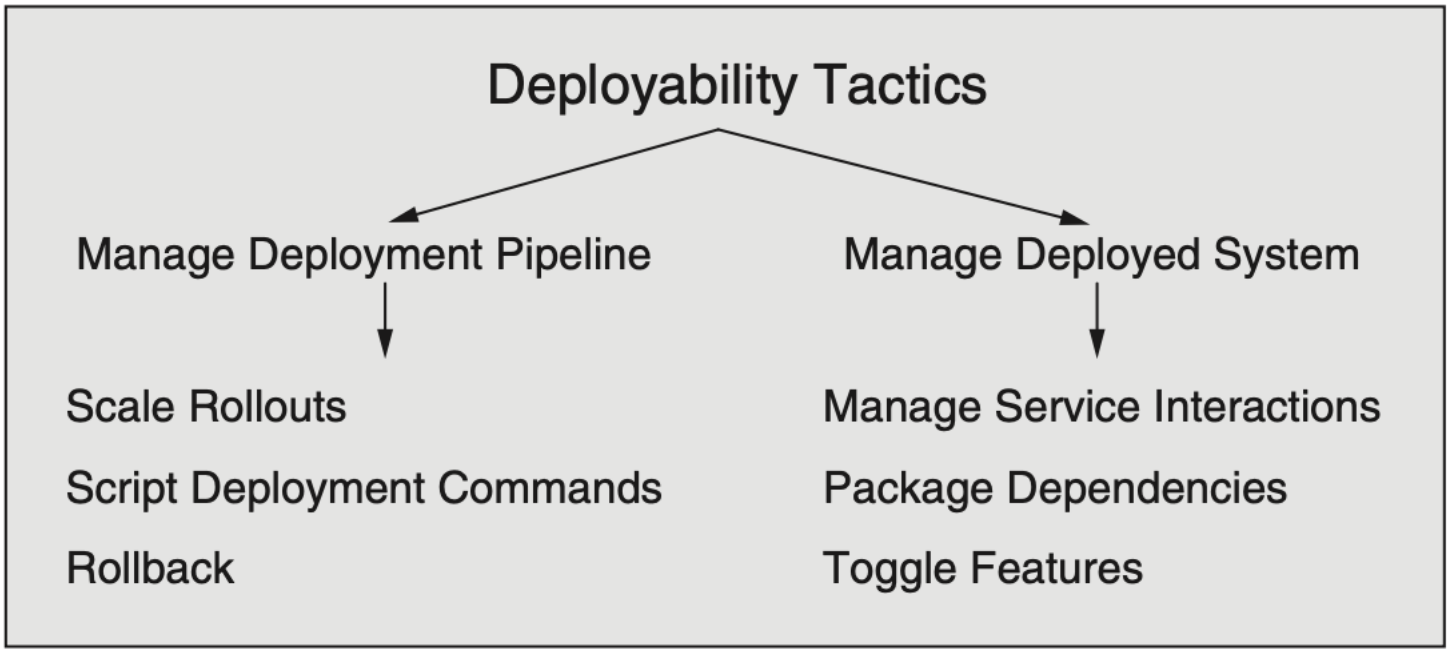


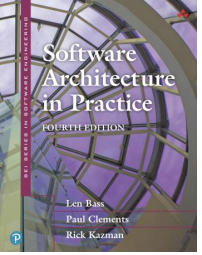
Goal of Deployability Tactics





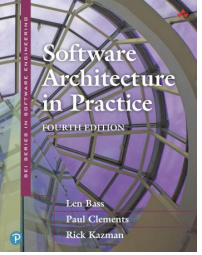
Deployability Tactics





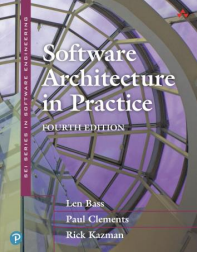
Manage Deployment Pipeline

- *Scale rollouts*. Rather than deploying to the entire user base, scaled rollouts deploy a new version of a service gradually, to subsets of the user population. By gradually releasing, the effects of new deployments can be monitored and measured and, if necessary, rolled back.
- *Roll back*. If it is discovered that a deployment has defects it can be "rolled back" to its prior state. Since deployments may involve multiple updates of multiple services, the rollback mechanism must be able to keep track of these, or reverse the consequences of any update, ideally in a fully automated fashion.
- *Script deployment commands*. Deployments are often complex and require many steps to be carried out and orchestrated precisely. For this reason, deployment is often scripted.



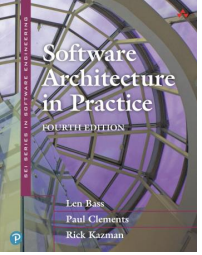
Manage Deployed System

- *Manage service interactions*. This tactic accommodates simultaneous deployment of multiple versions of services. The interactions between services need to be mediated so that version incompatibilities are avoided.
- *Package dependencies*. This tactic packages an element with its dependencies so that they get deployed together and versions of the dependencies are consistent.
- *Feature toggle*. Issues may arise after deploying new features. For that reason, we often integrate a “kill switch” (feature toggle) for new features. This automatically disables a feature at runtime, without forcing a new deployment.



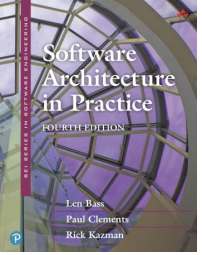
Tactics-Based Questionnaire for Deployability

Tactics Groups	Tactics Question	Supported? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Manage deployment pipeline	Do you **scale rollouts**, rolling out new releases gradually (in contrast to releasing in an all-or-nothing fashion)?				
	Are you able to automatically **roll back** deployed services if you determine that they are not operating in a satisfactory fashion?				
	Do you **script deployment commands** to automatically execute complex sequences of deployment instructions?				
Manage deployed system	Do you **manage service interactions** so that multiple versions of services can be safely deployed simultaneously?				
	Do you **package dependencies** so that services are deployed along with all of the libraries, OS versions, and utility containers that they depend on?				
	Do you employ **feature toggles** to automatically disable a newly released feature (rather than rolling back the newly deployed service) if the feature is determined to be problematic?				



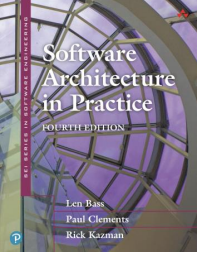
Microservice Pattern for Deployability

- The microservice architecture pattern structures the system as a collection of independently deployable services that communicate only via messages through service interfaces.
- There is no other form of interprocess communication allowed.
- Services are usually stateless, and (because they are developed by a relatively small team) are small—hence the term *microservice*.
- Service dependencies are acyclic. An integral part of this pattern is a discovery service so that messages can be appropriately routed.



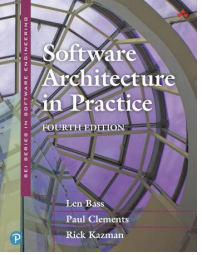
Microservice Pattern Benefits

- Benefits:
  - Time to market is reduced. Since each service is small and independently deployable, a modification to a service can be deployed without coordinating with other teams and services.
  - Each team can make its own technology choices, as long as the technology choices support message passing. This reduces errors due to incompatibilities.
  - Services are more easily scaled than coarser-grained applications. Since each service is independent, dynamically adding instances of the service is straightforward. In this way, the supply of services can be more easily matched to the demand.



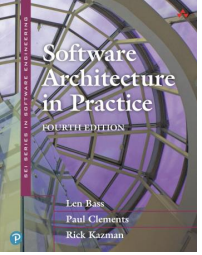
Microservice Pattern Tradeoffs

- Tradeoffs:
  - Overhead is increased, because all communication among services occurs via messages across a network. Furthermore, because of the dynamic nature of microservices, discovery services are heavily used, adding overhead.
  - Microservices are less suitable for complex transactions because of the difficulty of synchronization.
  - The freedom for every team to choose its own technology comes at a cost—the organization must maintain all those technologies and the required experience base.
  - Intellectual control of the total system may be difficult because of the large numbers of microservices.
  - Designing the services to have appropriate responsibilities and an appropriate level of granularity is a formidable design task.
  - To achieve the ability to deploy versions independently, the architecture of the services must be designed to allow for that deployment strategy.



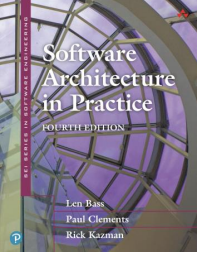
Patterns for Complete Replacement of Services

- You often need to replace all N running instances of a service with no reduction in QoS to the clients of the service; there must always be *N* instances of the service running.
- Two patterns for complete replacement are possible, both of which are realizations of the scale rollouts tactic: *blue/green* and *rolling upgrade*.



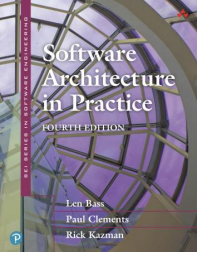
Blue/Green Pattern

- In a blue/green deployment, *N* new instances of a Service A are created (the green instances).
- After the *N* instances of new Service A are installed, the DNS server or discovery service is changed to point to the new version.
- Once it is determined that the new instances are working satisfactorily, then and only then are the *N* instances of original Service A removed.
- Before this cutoff point, if a problem is found in the new version, it is a simple matter of switching back to the original (the blue services) with little or no interruption.



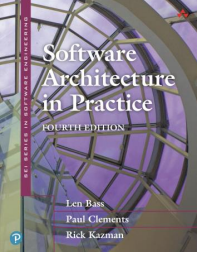
Rolling Upgrade Pattern

- A rolling upgrade replaces instances of Service A with instances of the new version one at a time (or a small fraction at a time). The steps of the rolling upgrade are as follows:
  - Allocate resources for a new instance of Service A.
  - Install and register the new version of Service A.
  - Begin to direct requests to the new version of Service A.
  - Choose an instance of the old Service A, allow it to complete any active processing,
  - Destroy that instance.
  - Repeat the preceding steps until all instances of the old version have been replaced.



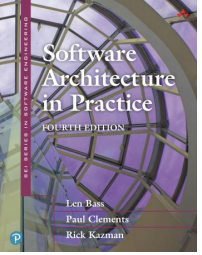
Complete Replacement of Services Patterns Benefits

- The benefit of these patterns is the ability to completely replace deployed versions of services without having to take the system out of service, thus increasing the system's availability.



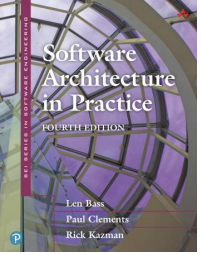
Complete Replacement of Services Patterns Tradeoffs

- The peak resource utilization for a blue/green approach is 2*N* instances, whereas the peak utilization for a rolling upgrade is *N* + 1 instances. In either case, resources to host these instances must be procured. With the widespread adoption of cloud computing, this tradeoff is less compelling but still present.
- Suppose there is an error in the new Service A. If you are using blue/green deployment, by the time you discover the error, all of the original instances may have been deleted and rolling back to the old version could take considerable time. In contrast, a rolling upgrade may allow you to discover an error in the new version of the service while instances of the old version are still available.



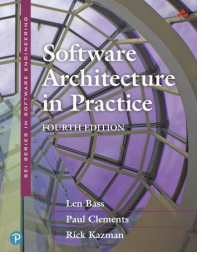
Complete Replacement of Services Patterns Tradeoffs

- If employ blue/green deployment, then at any time a client is either using new version or the old version. If you are using rolling upgrade, both versions are simultaneously active. This introduces two potential problems:
  - *Temporal inconsistency*. In a sequence of requests by Client C to Service A, some may be served by the old version of the service and some may be served by the new version. If the versions behave differently, this may cause Client C to produce erroneous, or at least inconsistent, results. (This can be prevented by using the manage service interactions tactic.)
  - *Interface mismatch*. If the interface to the new version of Service A is different from the old version, then invocations by clients that have not been updated to reflect the new interface will produce unpredictable results. This can be prevented by extending (but not modifying) the existing interface, and using the mediator pattern (see Chapter 7) to translate interface.



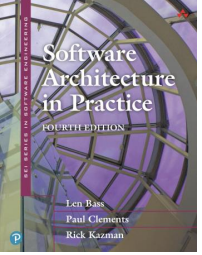
Patterns for Partial Replacement of Services

- Sometimes changing all instances of a service is undesirable.
- Partial-deployment patterns aim at providing multiple versions of a service simultaneously for different user groups.
- They are used for purposes such as quality control (canary testing) and marketing tests (A/B testing).



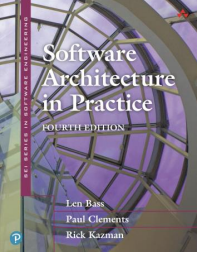
Canary Testing

- Before rolling out a new release, it is prudent to test it in the production environment, but with a limited set of users. Canary testing designates a small set of users who will test the new release.
- These "canary" users are routed to the appropriate version of a service. After testing is complete, users are all directed to either the new version or the old version, and instances of the deprecated version are destroyed.
- Rolling upgrade or blue/green deployment could be used to deploy the new version.



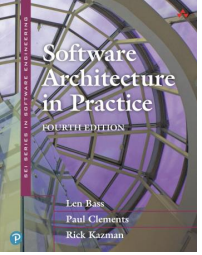
Benefits of Canary Testing

- Canary testing allows real users to “bang on” the software in ways that simulated testing cannot. This allows the organization deploying the service to collect “in use” data and perform controlled experiments with relatively low risk.
- Canary testing incurs minimal additional development costs, because the system being tested is on a path to production anyway.
- Canary testing minimizes the number of users who may be exposed to a serious defect in the new system.



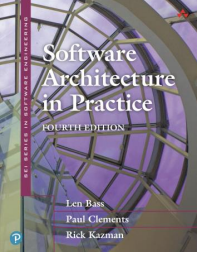
Tradeoffs of Canary Testing

- Canary testing requires additional up-front planning and resources, and a strategy for evaluating the results of the tests needs to be formulated.
- If canary testing is aimed at power users, those users have to be identified and the new version routed to them.



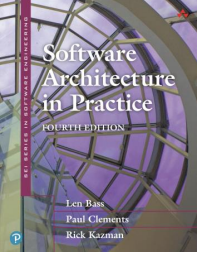
A/B Testing

- A/B testing is used by marketers to perform an experiment with real users to determine which of several alternatives yields the best business results. A small number of users receive a different treatment from other users.
- The “winner” would be kept, the “loser” discarded, and another contender designed and deployed.



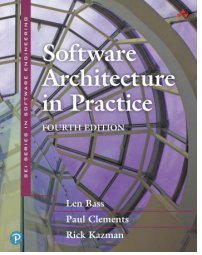
Benefits of A/B Testing

- A/B testing allows marketing and product development teams to run experiments on, and collect data from, real users.
- A/B testing can allow for targeting of users based on an arbitrary set of characteristics.



Tradeoffs of A/B Testing

- A/B testing requires the implementation of alternatives, one of which will be discarded.
- Different classes of users, and their characteristics, need to be identified up front.



Summary

- *Deployability* is a property of a software system indicating that it may be deployed predictably.
- Architects are concerned with the degree to which architectures are: *granular, controllable, and efficient.*
- This requires that the architect pays attention to all phases of the deployment pipeline and designs appropriate levels of automation in the pipeline.