Web Services, DevOps, and Deployment

UNSW COMP9243 lecture, 2/5/2017

Dr. Ingo Weber | Principal Research Scientist & Team Leader

Agenda

- Web services and SOA
- RESTful services
- DevOps Overview
- Microservices
- Continuous Deployment

Some material is based on Dr. Ihor Kuz's material from COMP9243, other material from COMP9322 lecture slides, prepared by myself, Dr. Helen Hye-young Paik, and Dr. Sherif Sakr. Some material is adapted from one of the textbook Web Services: Concepts, Architectures and Applications and is thus under Copyright 2003 Gustavo Alonso, ETH Zurich and/or Copyright 2004 Springer-Verlag Berlin Heidelberg. Some content is based on *DevOps: A Software Architect's Perspective*, L. Bass, I. Weber, L. Zhu, Addison-Wesley Professional, 2015. Other references are given throughout the slides.

Motivation for Web services and SOA

- Integration of complex, distributed IT systems was complex problem
  - Example: Java remote procedure calls only possible between same version of Java
- Need to abstract from implementation platforms, model, methodologies and tools

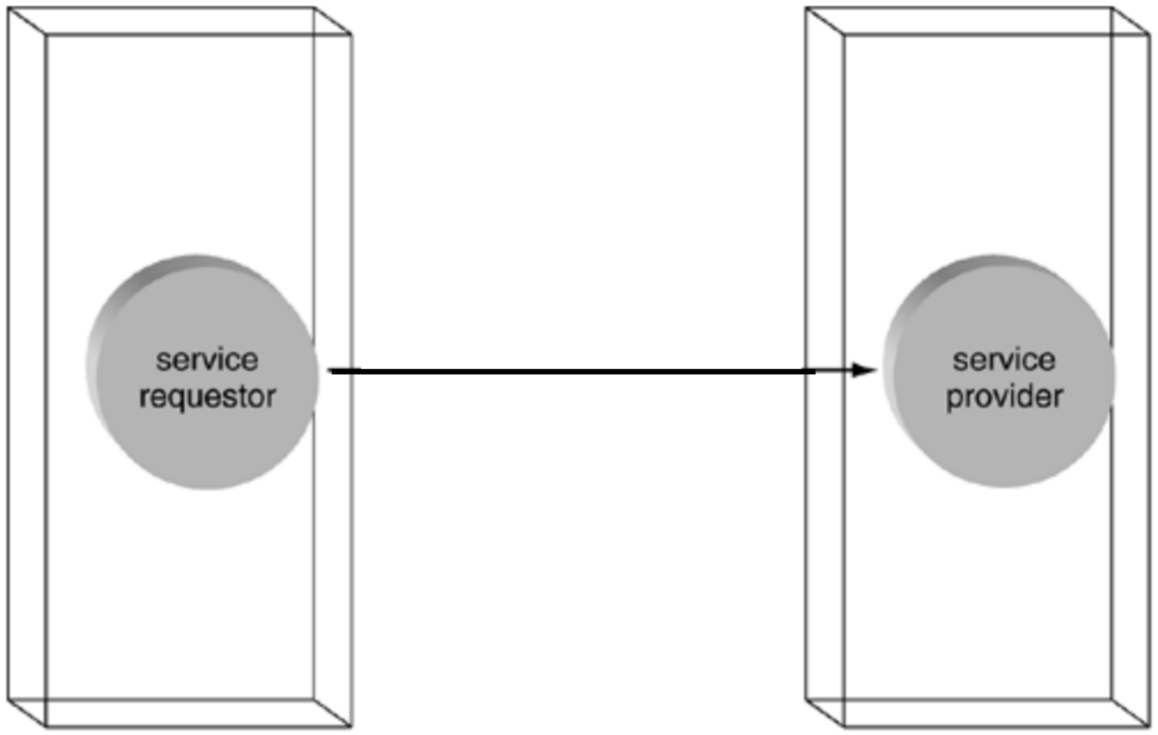
→*Service-orientation* (SO): approach & principle behind SO Architecture and SO Computing

- SO means to think, model, and implement software systems in terms of services

What is a Web service?

- A Web service is a component wrapped behind a standardized interface
  - Typical technologies: SOAP/WSDL or RESTful
  - Loosely-coupled: the services are independent of each other, heterogeneous, distributed
  - Message based: interaction can be through message exchanges rather than through direct calls (unlike CORBA, RPC, etc.)
- Services traditionally were large-scale
  - Did not deliver all promises
  - Overhead too big for convenient use in JavaScript or other lightweight environments

Service orientation: everything is a service

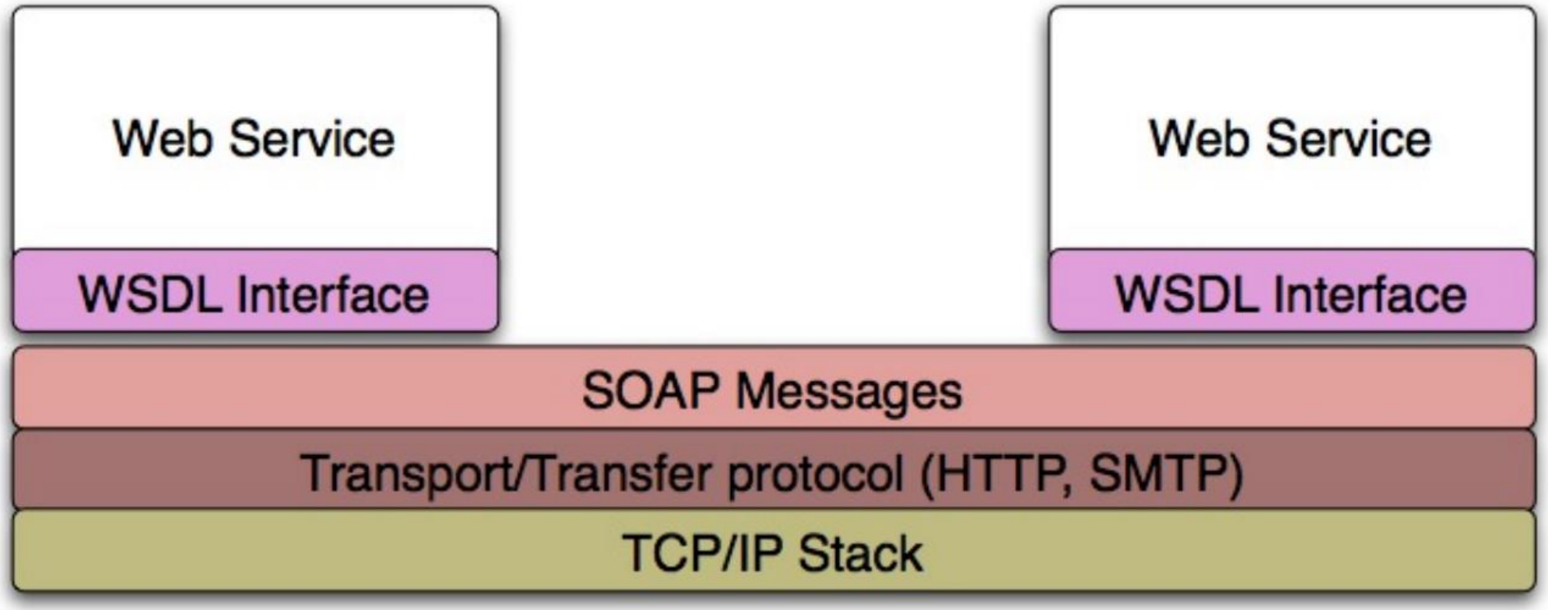


Web services

- Key concepts:
  - Allow applications to share data and invoke capabilities from other applications
    - Without regard to how those applications were built, what operating system or platform they run on, and what devices are used to access them
  - Can be called across platforms and operating systems, regardless of programming language
- Key facts:
  - A standardised way of application-to-application communication based on open XML standards (i.e., SOAP, WSDL and UDDI) and over a standard Internet protocol / backbone
  - Unlike traditional client/server models, Web services do not require GUI, HTML or browser

Brief overview: SOAP/WSDL

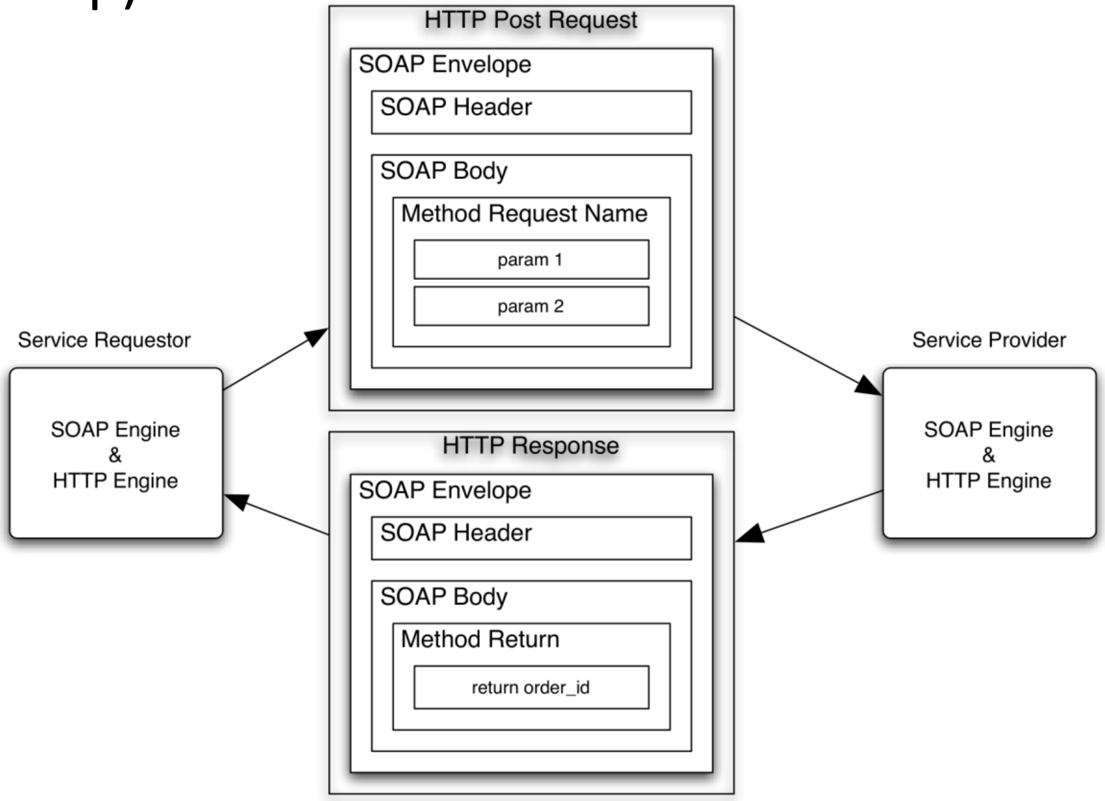
- The “original” Web service standards
- WSDL: Web-services Description Language (www.w3.org/TR/wsdl)
- SOAP (www.w3.org/TR/soap)



Developing Enterprise Web Services: An Architect’s Guide, S. Chatterjee and J. Webber, Prentice Hall, p72

Brief overview: SOAP/WSDL

- The “original” Web service standards
- WSDL: Web-services Description Language (www.w3.org/TR/wsdl)
- SOAP (www.w3.org/TR/soap)



What is SOA?

- SOA = Service-Oriented Architecture
  - (Almost) all components are services
  - Architecture: SOA is intended as a way to build applications and follows on previous ideas such as software bus, IT backbone, or enterprise bus
- Major difference to conventional middleware: standardization efforts through the W3C
  - Implementation technology independence
- Based on many well-known concepts, with some new challenges and some “reinventing the wheel”

SOA vs. Web services

- Web services are about:
  - Interoperability
  - Standardization
  - Integration across heterogeneous, distributed systems
- Service Oriented Architectures are about:
  - Large-scale software design
  - Software Engineering
  - Architecture of distributed systems
- SOA introduced some radical changes to software:
  - Language independence (what matters is the interface)
  - Event-based interaction (asynchronous; synchronous still supported)
  - Message-based exchanges (RPC-like calls are an option, not must)
  - Composition of services
- SOA is possible but more difficult without Web services

Service-Orientation Design Principles

- **Standardized Service Contract:** the public interfaces of a services make use of contract design standards. (Contract: WSDL in WS*)
- **Service Loose Coupling:** to impose low burdens on service consumers (coupling ~ degree of dependency)
- **Service Abstraction:** “to hide as much of the underlying details of a service as possible”
- **Service Reusability:** services contain agnostic logic and “can be positioned as reusable enterprise resources”
- **Service Autonomy:** to provide reliable and consistent results, a service has to have strong control over its underlying environment

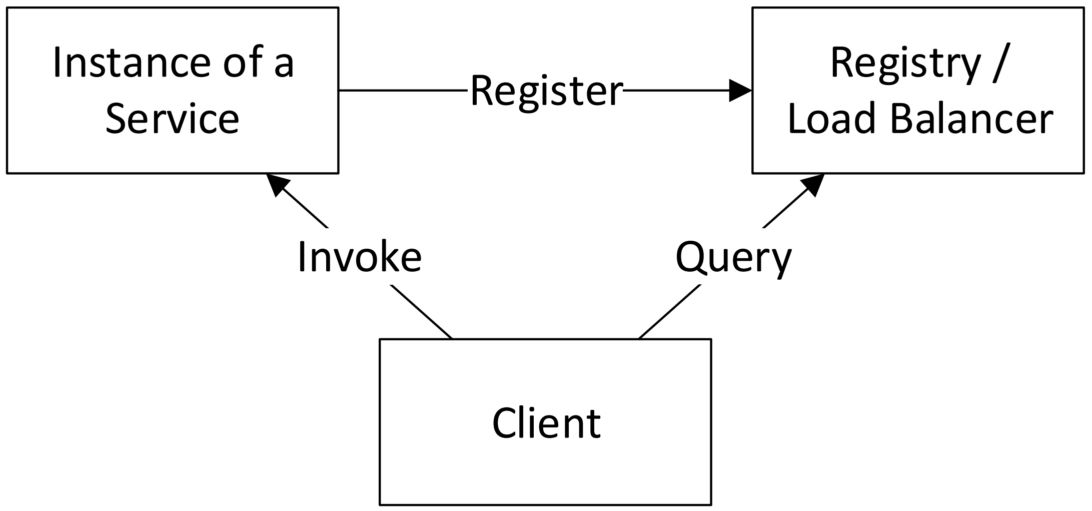
Based on SOA Principles of Service Design, Thomas Erl, Prentice Hall, 2007, http://serviceorientation.com/serviceorientation.
Summary: I. Weber, Semantic Methods for Execution-level Business Process Modeling. Springer LNBIP Vol. 40, 2009.

Service-Orientation Design Principles

- **Service Statelessness:** services should be “designed to remain stateful only when required.”
- **Service Discoverability:** “services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.”
- **Service Composability:** “services are effective composition participants, regardless of the size and complexity of the composition.”
- **Fundamental requirement – interoperability of services:** “...stating that services must be interoperable is just about as evident as stating that services must exist.”

Based on SOA Principles of Service Design, Thomas Erl, Prentice Hall, 2007, http://serviceorientation.com/serviceorientation.
Summary: I. Weber, Semantic Methods for Execution-level Business Process Modeling. Springer LNBIP Vol. 40, 2009.

Service discovery



- When an instance of a service is launched, it registers with a registry/load balancer
- When a client wishes to utilize a service, it gets the location of an instance from the registry/load balancer
- Netflix Eureka is an open source registry/load balancer

Agenda

- Web services and SOA
- <u>RESTful services</u>
- DevOps Overview
- Microservices
- Continuous Deployment

REST philosophy

- REST is an *architectural style* of networked systems (not a protocol – not a specification)
- Objective: Expose resources on a networked system (the Web)
- Principles:
  - Resource Identification using a URI (Uniform Resource Identifier)
  - Unified interface to retrieve, create, delete or update resources
  - REST itself is not an official standard or even a recommendation. It is just a "design guideline"

Reference: RESTful Web Services, L. Richardson and S. Ruby, O'Reilly.

REST philosophy: resources

- A resource is a thing that:
  - is unique (i.e., can be identified uniquely)
  - has at least one representation,
  - has one or more attributes beyond ID
  - has a potential schema, or definition
  - can provide context
  - is reachable within the addressable universe
- Examples:
  - Web Site, resume, aircraft, song, transaction, employee, application, Blog posting, printer, …

Origins of REST

- REST is an acronym standing for *Representational State Transfer*
  - First introduced by Roy T. Fielding in his PhD dissertation ”Architectural Styles and the Design of Network-based Software Architectures”
  - He focused on the rationale behind the design of the modern Web architecture and how it differs from other architectural styles.
- REST is client-server where a *representation* of the resource is exposed to the client application
- The representation of resources places the client application in a *state*
- The state is evolving with each resource representation through traversing hyperlinks

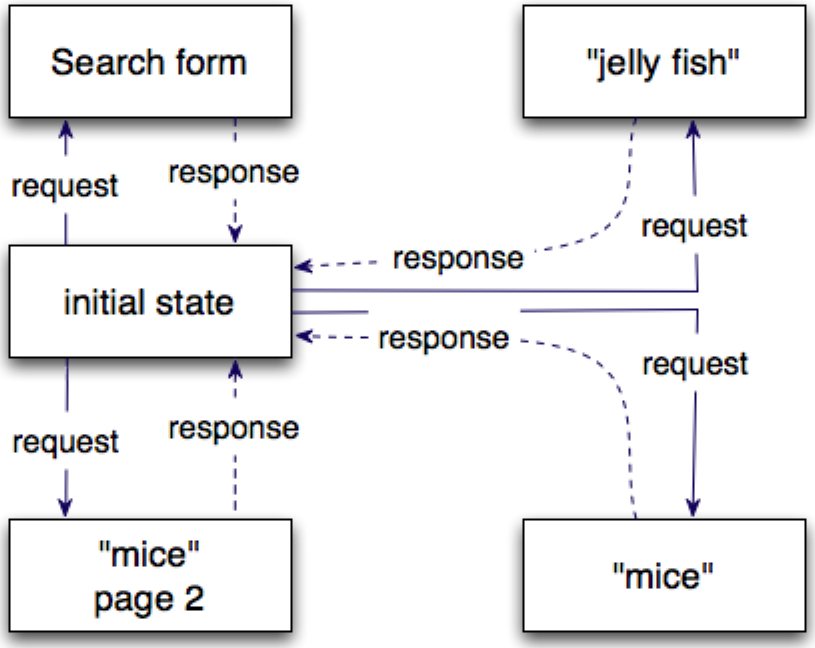
REST principles: Resource Identification & Addressability

- Resources are identified by a URI (Uniform Resource Identifier)
  - http://www.example.com/software/release/1.0.3.tar.gz
- A resource has to have at least one URI
- Most known URI types are URL and URN
  - URN (Uniform Resource Name)
    - urn:isbn:0-486-27557-4
  - URL (Uniform Resource Locator)
    - http://www.google.com.au
- Every URI designates exactly one resource
- Make systems addressable:
  - An application is “addressable” if it exposes its data set as resources (i.e., usually an infinite number of URIs)
  - E.g., Google search: http://www.google.com.au/search?q=unsw

REST principles: Statelessness

Focus: server state

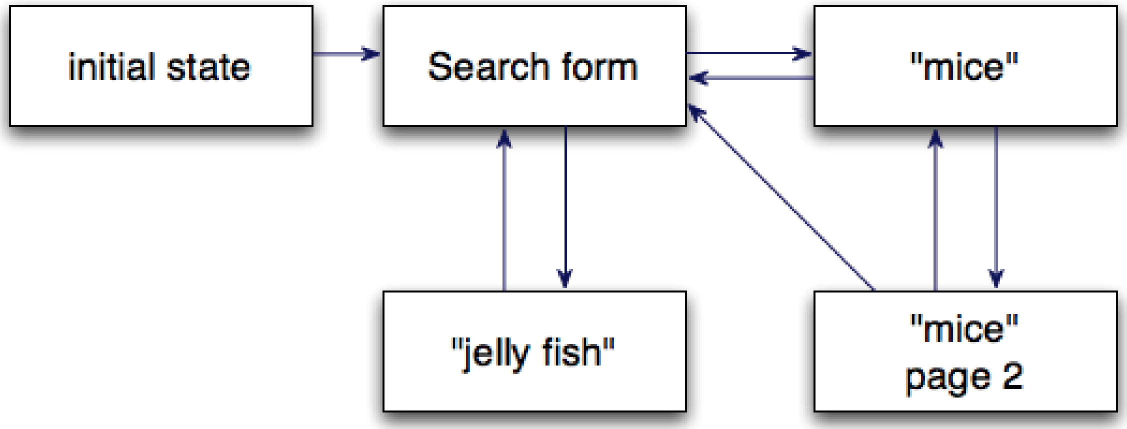
Stateless means every HTTP request happens in a complete isolation. When a client makes a HTTP request, it includes necessary information for the server to fulfil the request



A Stateless Search Engine: Page 88 RESTful Web Services

REST principles: Statelessness

In contrast, some Web sites that expects you to make requests in a certain order: A, B, then C. (i.e., it gets confusing when you make request B a second time instead of moving onto request C)



A Statelful Search Engine: Page 88 RESTful Web Services

REST principles: Statelessness

- REST principle says: URI needs to contain the state within it, not just a key to some state stored on the server
  - RESTful service requires that the client state stays on the client side
  - Client transmits its state to the server for every request that needs it
    - E.g.: https://www.google.com.au/search?q=cloud&ie=utf-8&oe=utf-8&client=firefox-b-ab&...
  - Server supports the client in navigating the system by sending ’next links’ the client can follow
    - E.g.: https://www.google.com.au/search?q=cloud&...&start=10...
  - Server does not keep track of the state on behalf of a client

REST principles: Statelessness

- Client / Application State vs. Resource State
  - What counts as ’state’ exactly then? Think Flickr.com: would statelessness mean that I have to send every one of my pictures along with every request to flickr.com?
- Two kinds of states
  - Resources, like photos
  - Client state: the ‘application’ that accesses resources
- Resource state lives on the server (i.e., photos are stored on the server)
- Client application state should be kept off the server
- Statelessness in REST applies to the client application state
- Resource state is the same for every client and its state is managed on the server.

REST principles: Resource Representations

- A resource needs a representation for it to be sent to the client
  - a representation of a resource - some data from the ’current state’ of a resource (i.e., a list of open bugs)
  - a list of open bugs – either in XML, JSON, HTML, comma-separated values, printer-friendly format, ...
  - a representation of a resource may also contain metadata about the resource (e.g. for books: the book itself + metadata such as cover-image, reviews, stock-level, etc.)
- Representation can flow the other way too: a client sends a ‘representation’ of a new resource and the server creates the resource

REST principles: Uniform Interface

- REST Uniform Interface Principle uses 4 main HTTP methods on resources
  - **PUT**: Update a resource (existing URI).
  - **GET**: Retrieve a representation of a resource.
  - **POST***: Create a new resource or modify the state of a resource. POST is a read-write operation and may change the state of the resource and provoke side effects on the server. Web browsers warn you when refreshing a page generated with POST.
  - **DELETE**: Clear a resource, afterwards the URI is no longer valid
  - HEAD and OPTIONS
- Similar to the CRUD (Create, Read, Update, Delete) databases operations

*POST: a debate about its exact best-practice usage – some people claim ‘create’ should be done via PUT.

REST principles: Safety and Idempotence

REST Uniform Interface, if properly followed, gives you two properties:

- Safe operations
  - Read-only operations: the operations on a resource do not change any server state. The client can call the operations 10 times, it has no effect on the server state. (Server state may change during the 10 calls though, for other reasons.)
- Idempotent operations
  - Operations that have the same “effect” whether you apply them once or more than once. An effect here may well be a change of server state. An operation on a resource is idempotent if making one request is the same as making a series of identical requests.

REST principles: Safety and Idempotence

- Math analogy:
  - multiplying a number by zero is *idempotent*: 4x0x0x0x0 is the same as 4x0
  - multiplying a number by one *is both safe and idempotent*: 4x1x1x1x1 is the same as 4x1 (idempotence) but also it does not change 4 (safe)
- GET, HEAD, OPTIONS: safe (and thus idempotent)
- PUT: idempotent
  - If you create a resource with PUT, and then resend the PUT request, the resource is still there and it has the same properties you gave it when you created it. Same for update.
- DELETE: idempotent
  - If you delete a resource with DELETE, it’s gone. You send DELETE again, it is still gone
- POST: neither

REST principles: Safety and Idempotence

- Why Safety and Idempotence matter
  - The two properties let a client make reliable HTTP requests over an unreliable network.
  - Your GET request gets no response? Retry, it’s safe
  - Your PUT request gets no response? Retry – even if your earlier one got through, your second PUT will have no side-effect
- Many applications misuse HTTP interface, e.g.,
  - GET https://api.del.icio.us/posts/delete
  - GET www.example.com/registration?new=true&name=aaa&ph=123
- Many applications expose unsafe operations as GET, and there are many uses of the POST method which are neither safe nor idempotent. Repeating them has consequences ...

Agenda

- Web services and SOA
- RESTful services
- <u>DevOps Overview</u>
- Microservices
- Continuous Deployment

Content based in part on *DevOps: A Software Architect's Perspective*, L. Bass, I. Weber, L. Zhu, Addison-Wesley Professional, 2015. Some slides courtesy of Len Bass

Why DevOps?

- Developers (Devs) and operators (Ops) don‘t always pursue the same goals

What problem is DevOps trying to solve?

- Poor communication between Dev and Ops
- Opposing goals
  - Devs want to push new features
  - Ops want to keep the system available

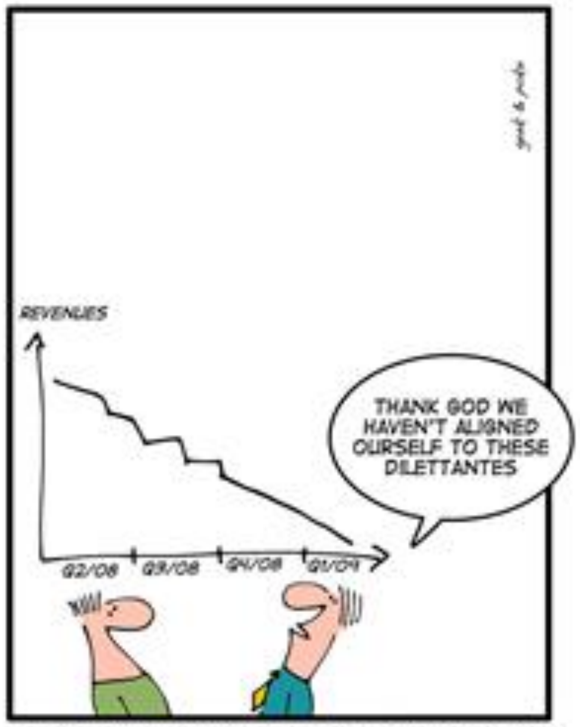
  →Leads to slow release schedule
- Limited capacity of operations staff
- Devs have limited insight into operations
- Why companies care

  "IBM has gone from spending about 58% of its development resources on innovation to about 80%."

  http://devops.com/blogs/ibms-devops-journey/?utm_content=12855120

DevOps motivation

- Organizations want to reduce time to market for new features, without sacrificing quality
  - Requires business-IT alignment
- DevOps practices will influence…
  - the way you organize teams
  - the way you build systems
  - even the structure of the systems that you build
- Unlikely that you'll be able to "throw your final version over the fence" and let operations worry about running it!

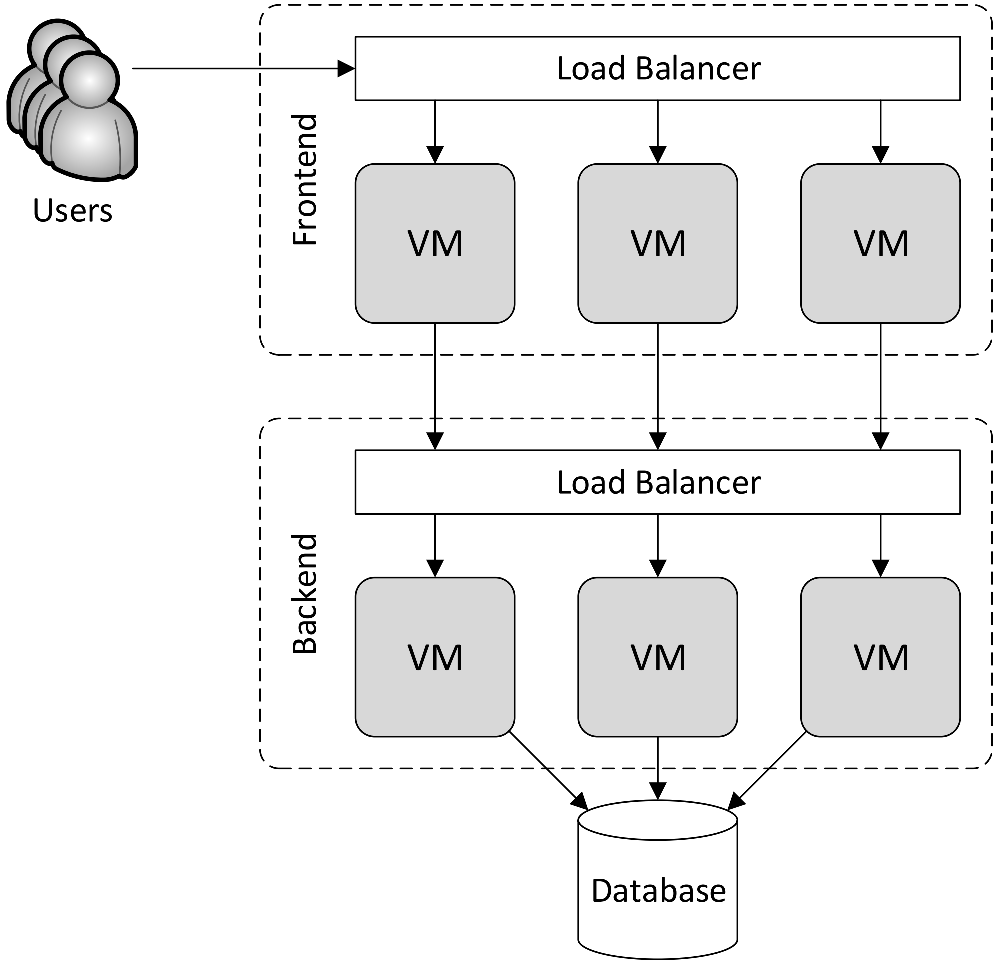


IT-FOLKS DON'T LIKE BUSINESS-IT ALIGNMENT

DevOps motivation

- DevOps is about...
  - bringing “agile” methods to operations
  - encouraging collaboration between development and operations staff, get them talking
  - Formally: shared goals and teams of Devs and Ops
  - Informally: beer & chips

Moving to the Cloud: Sample architecture

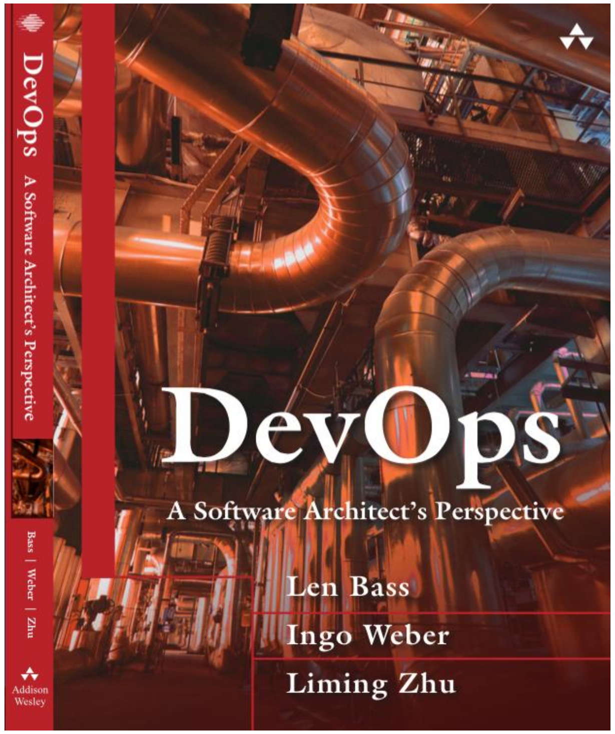


What is DevOps?

- Definition

  "DevOps is a set of practices intended to reduce the **time between committing** a change to a system and the change being **placed into normal production**, while ensuring **high quality**."

- Source:
  - http://nicta.info/devops
  - Release date: June 2015



What does that mean?

- Quality of the code must be high
  - Testing
- Quality of the build & delivery mechanism must be high
  - Automation & more testing
- Time is split:
  - From commit to deployment to production
  - From deployment to acceptance into normal production
- Goal-oriented definition
  - May use agile methods, continuous deployment (CD), etc.
  - Likely to use tools
- Achieving it starts before committing

DevOps Practices (1/2)

Requirements	Development	Build	Testing	Deployment	Execution
•Treat Operations personnel as first class stakeholders •Get their input when developing requirements	•Small teams •Limited coordination •Unit tests	•Build tools •Supports continuous integration	•Automated testing •Integration testing	•Deployment tools •Supports continuous deployment	•Monitoring •Responding to error conditions

- Treat Ops as first-class citizens throughout the lifecycle – e.g., in requirements elicitation
  - Many decisions can make operating a system harder or easier
  - Logging and monitoring to suit Ops
- Make Dev more responsible for relevant incident handling
  - Shorten the time between finding and repairing errors

DevOps Practices (2/2)

- Use continuous deployment, automate everything
  - Commits trigger automatic build, testing, deployment
- Enforce deployment process is used by all
  - No ad-hoc deployments
  - Ensures changes are traceable
- Develop infrastructure code with the same set of practices as application code
  - “Infrastructure as Code” : using IaaS APIs, etc., to automate creation of environments
  - Misconfiguration can derail your application
  - Ops scripts are traditionally more ad-hoc

Agenda

- Web services and SOA
- RESTful services
- DevOps Overview
- <u>Microservices</u>
- Continuous Deployment

Content based in part on *DevOps: A Software Architect's Perspective*, L. Bass, I. Weber, L. Zhu, Addison-Wesley Professional, 2015. Some slides courtesy of Len Bass

DevOps consequences

Speed up deployment through minimizing synchronous coordination among development teams.

- Synchronous coordination, like a meeting, adds time since it requires
  - Ensuring that all parties are available
  - Ensuring that all parties have the background to make the coordination productive.
  - Following up to decisions made during the meeting.

DevOps consequences

- Keep teams relatively small
  - Amazon's "two pizza rule": no team should be larger than can be fed with two pizzas
  - Advantages: make decisions quickly, less coordination overhead, more coherent units
- Team size becomes a major driver of the overall architecture:
  - Small teams develop small services
  - Coordination overhead is minimized by channeling most interaction through service interfaces:
    - Team X provides service A, which is used by teams Y and Z
    - If changes are needed, they are communicated, implemented, and added to the interface.

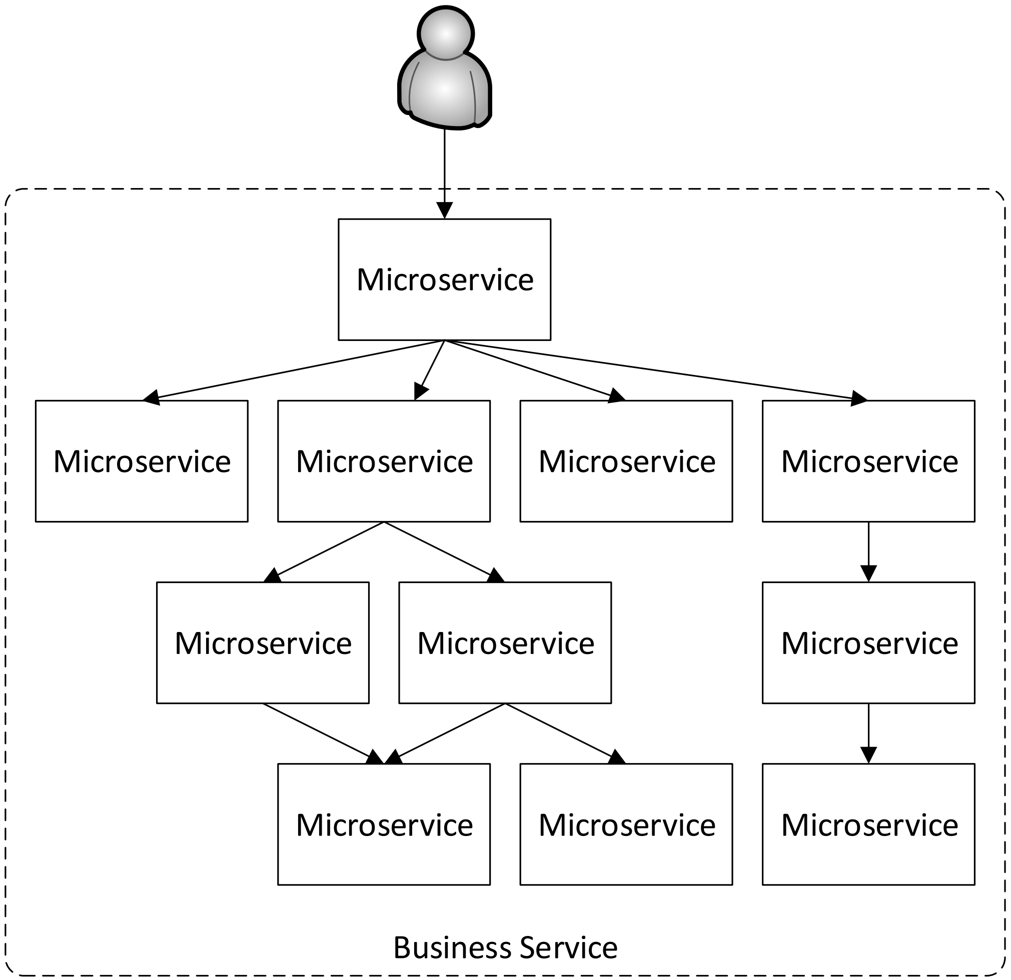
Microservice Architecture

- Microservice: small service that does one thing only, but does that well
  - Example: Atlassian BlobStore
- Used in practice by organizations that adopted (or invented) many DevOps practices
  - Amazon, LinkedIn, Google, …
- Each service provides small amount of functionality
- Total system functionality comes from composing multiple services

Amazon design rules

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed:
  - no direct linking, no direct reads of another team's data store,
  - no shared-memory model, no back-doors whatsoever.
  - The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they[services] use.
- All service interfaces, without exception, must be designed from the ground up to be externalizable.

Microservice Architecture



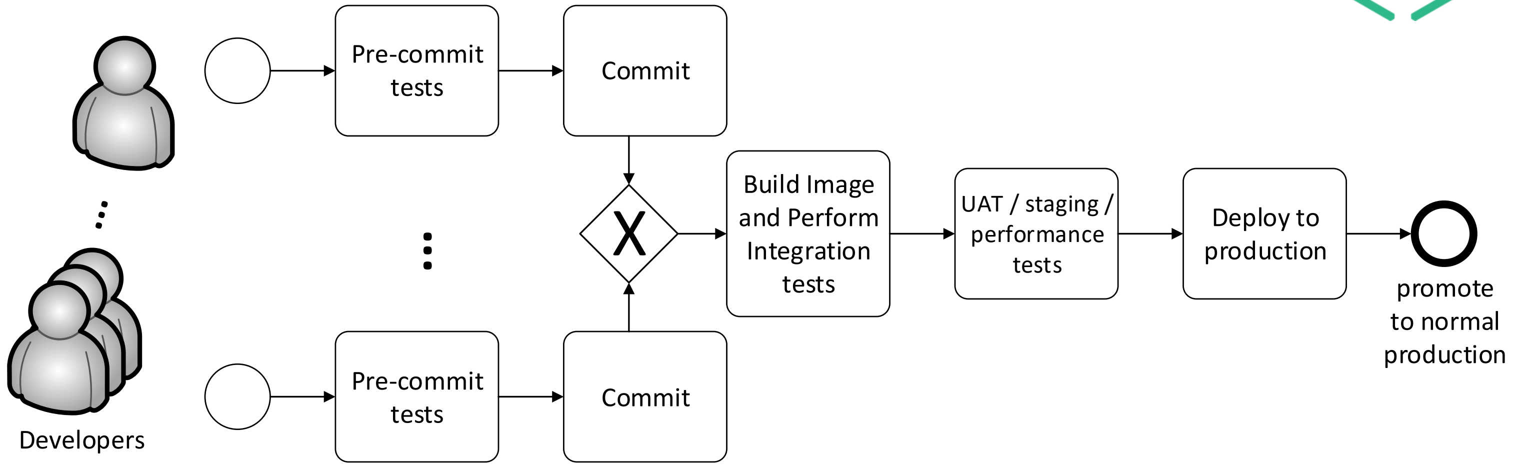
- Each user request is satisfied by some sequence of services
- Most services are not externally available
- Each service communicates with other services through service interfaces
- Service depth may be 70, e.g., LinkedIn

Agenda

- **Web services and SOA**
- **RESTful services**
- **DevOps Overview**
- **Microservices**
- **<u>Continuous Deployment</u>**

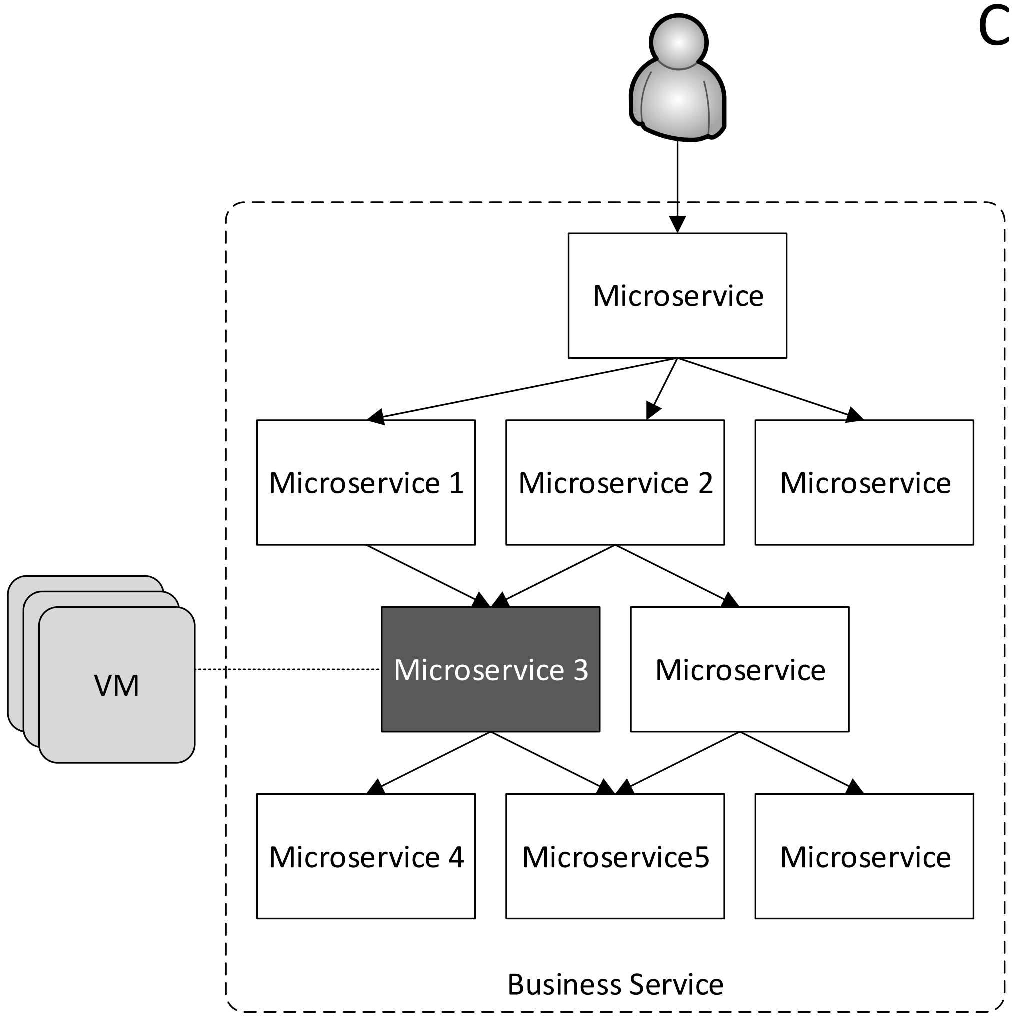
Content based in part on *DevOps: A Software Architect's Perspective*, L. Bass, I. Weber, L. Zhu, Addison-Wesley Professional, 2015. Some slides courtesy of Len Bass

Deployment pipeline


Pre-commit tests
Commit
Build Image and Perform Integration tests
UAT / staging / performance tests
Deploy to production
promote to normal production
Pre-commit tests
Commit
Developers

- Developer wants to commit code
- Pre-commit tests are executed locally. If successful:
- Code is committed
- Committed code is compiled, Unit tests are run. If successful:
- Code is built & packaged
    - Result can be a machine image or template (assuming virtualization). If successful:
- Integration tests are run. If successful:
- Acceptance / performance tests are run. If successful:
- The new service is deployed to production
- **All gates from one phase to the next are automatic – else continuous integration / delivery**

Deployment is not trivial



Challenges

- 24/7 availability is base requirement
- Microservice 3 to be replaced with new version
  - Multiple VMs for it
- Might change how M1 / M2 can use it
- Might change how it uses M4 / M5

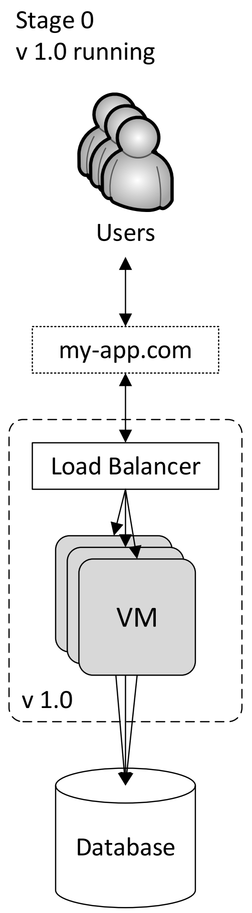
Deployment goal and constraints

- Goal: move from current state (*N* instances of version *A* of a service) to new state (*N* instances of version *B* of a service)
- Constraints:
  - Any development team can deploy their service at any time – no synchronization among development teams
  - It takes time to replace one instance of version *A* with an instance of version *B* (order of minutes)
  - Service to clients must be maintained while the new version is being deployed (24/7 availability)

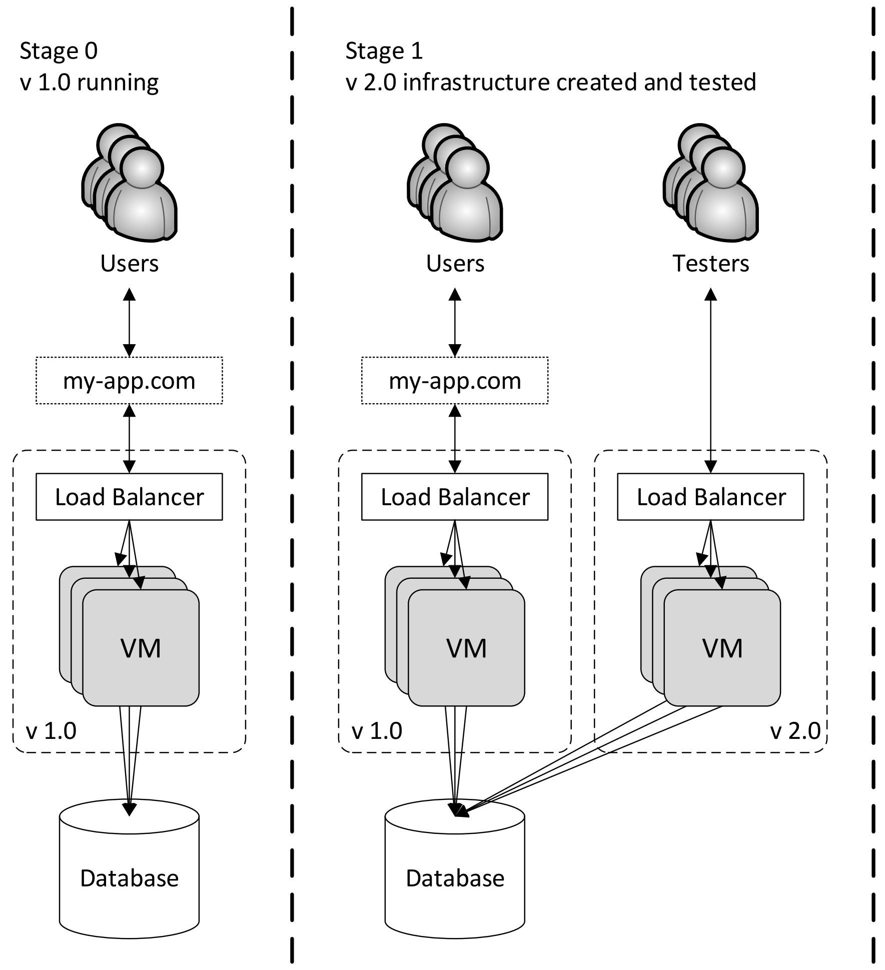
Deployment strategies

- Two basic all-or-nothing strategies:
  - Big Flip (or Blue/Green) Deployment
    - leave *N* VMs with version *A* as they are, allocate and provision *N* VMs with version *B,* switch to version *B*; once stable, release VMs with version *A*.
  - Rolling Upgrade
    - allocate a new VM with version *B*, release one VM with version *A*. Repeat *N* times.
- Other deployment topics
  - Partial strategies (canary testing, A/B testing, ...). Here focusing on all-or-nothing deployment.
  - Rollback
  - Packaging services into machine images

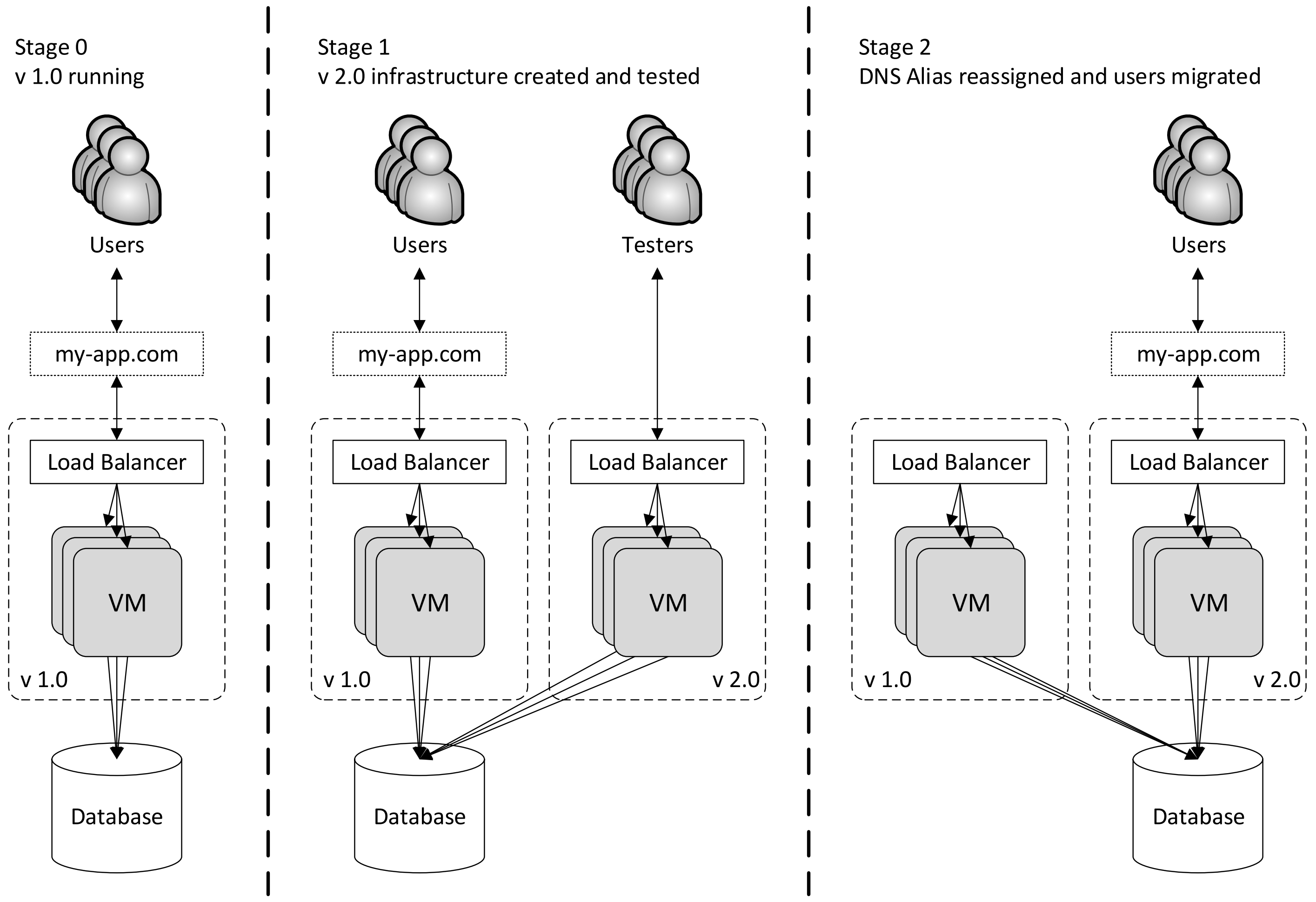
Blue/Green deployment (1/3)



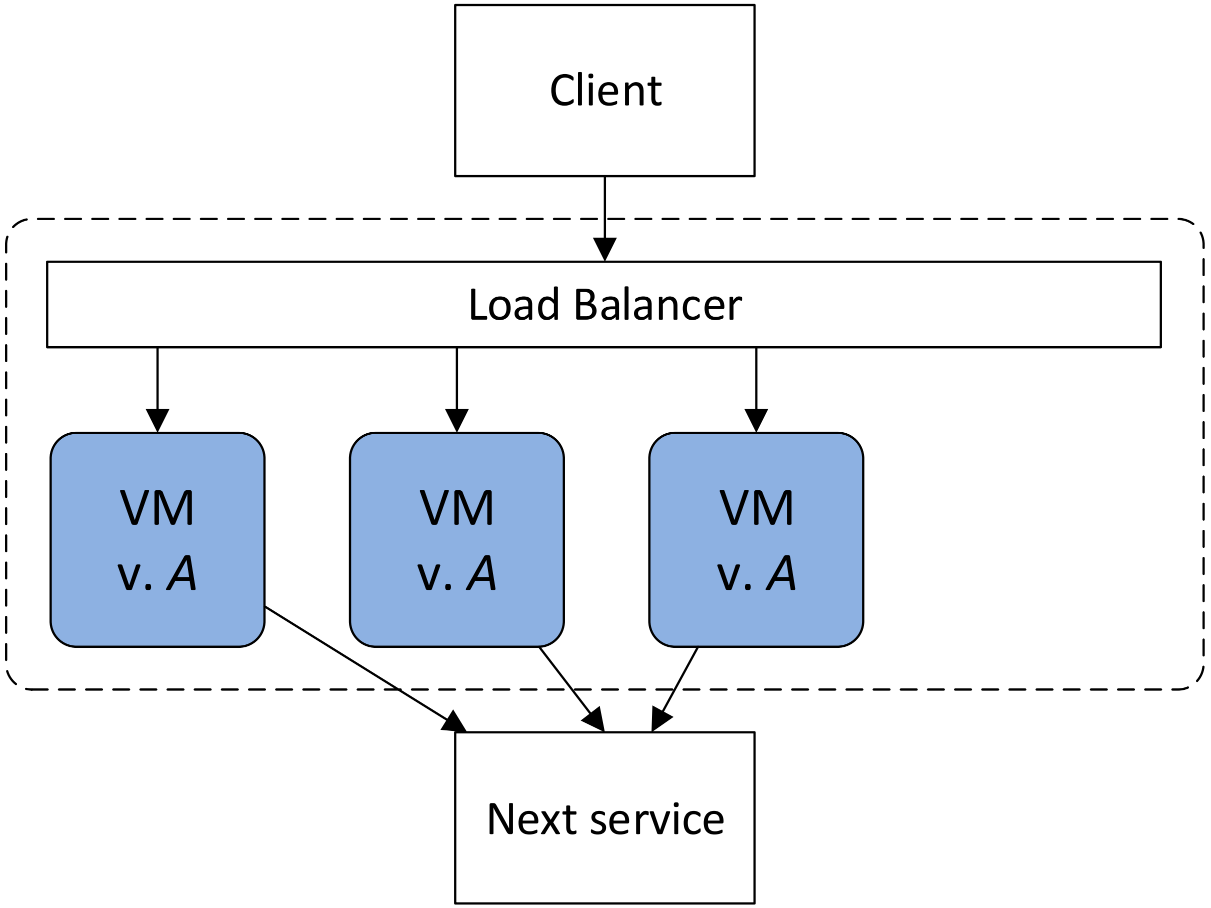
Blue/Green deployment (2/3)



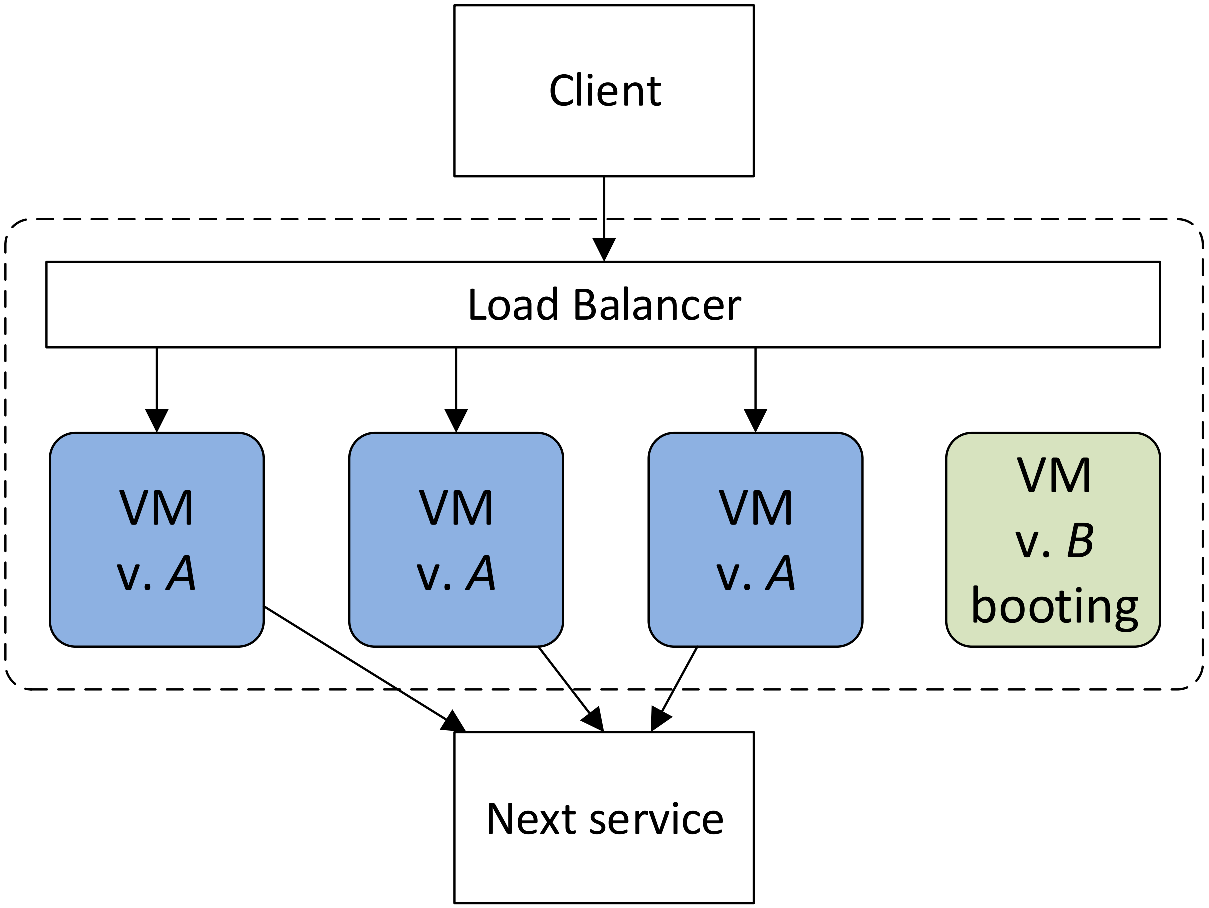
Blue/Green deployment (3/3)



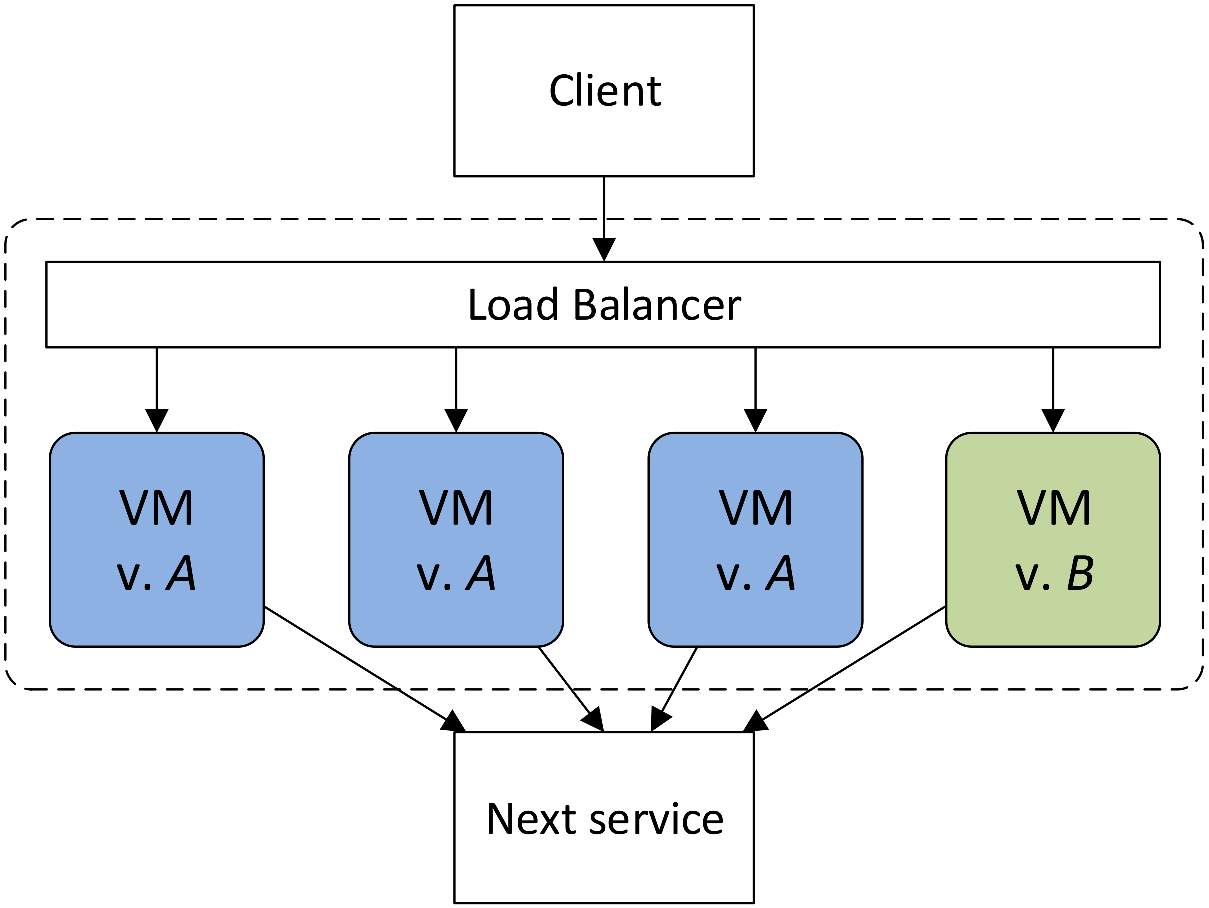
Rolling Upgrade



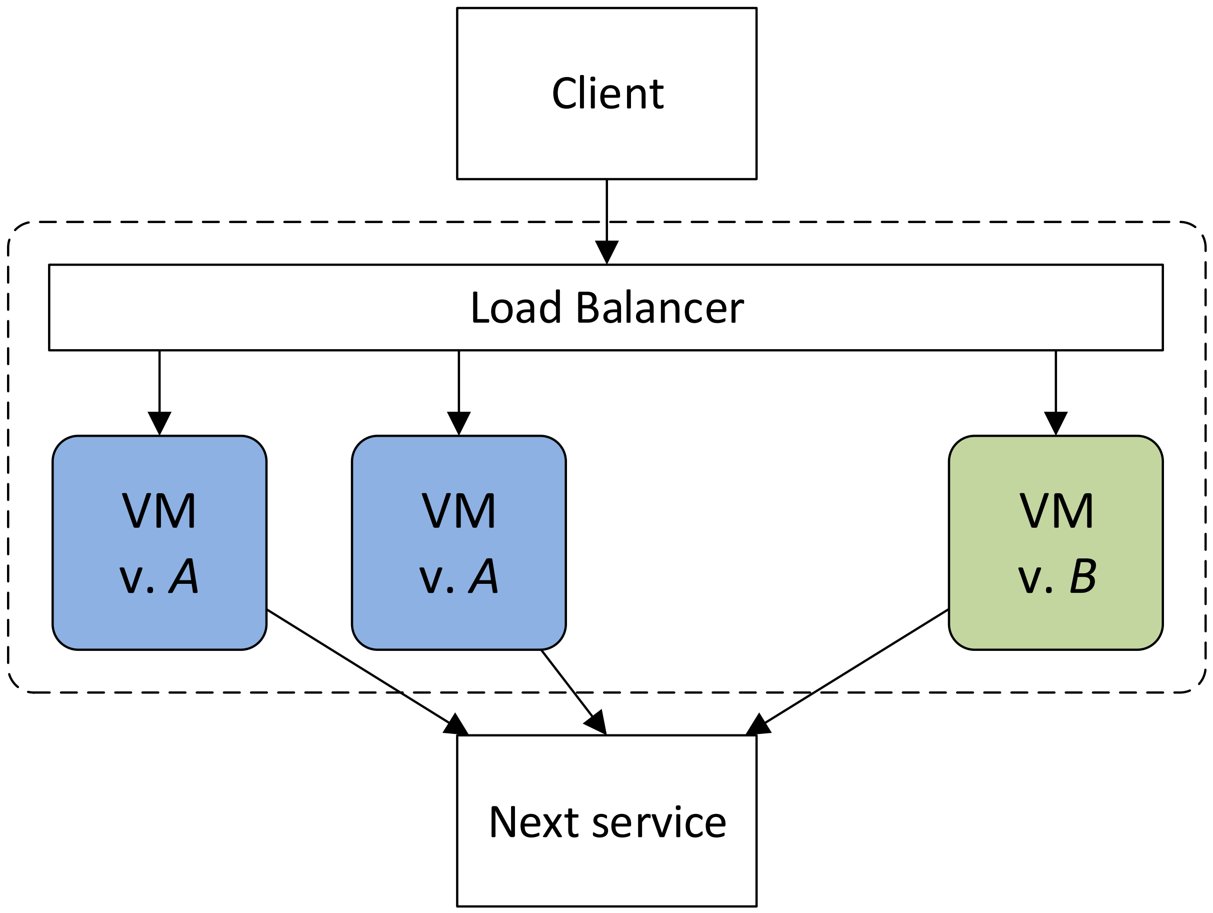
Rolling Upgrade



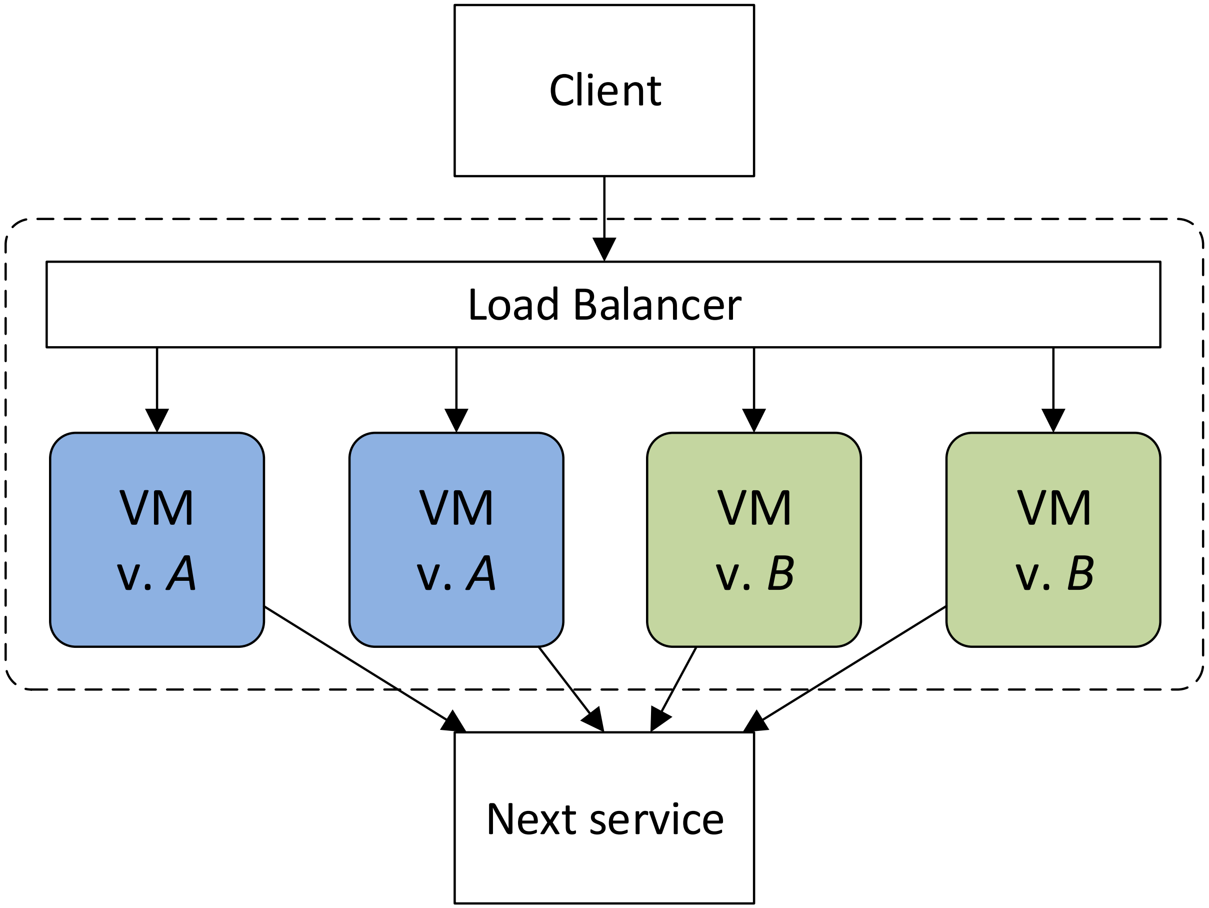
Rolling Upgrade



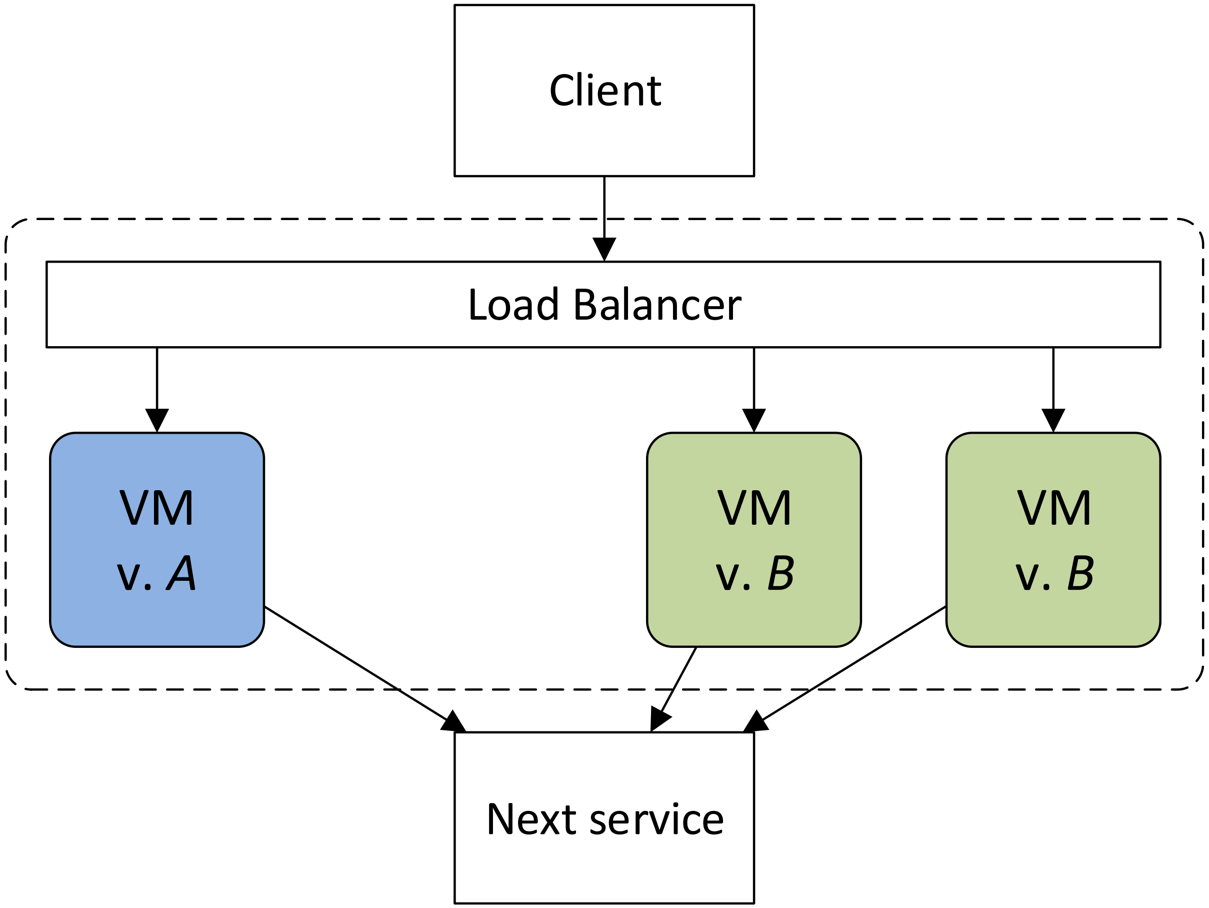
Rolling Upgrade



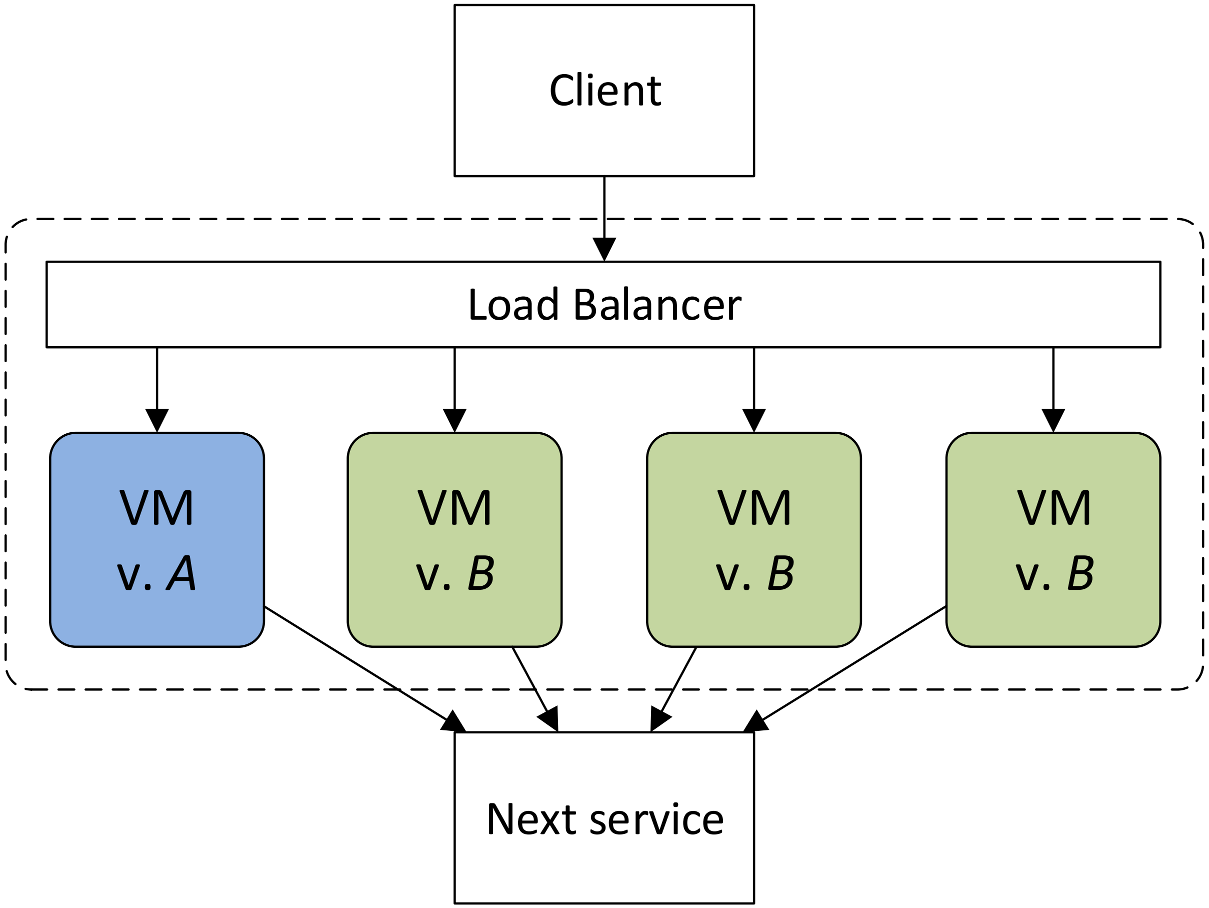
Rolling Upgrade



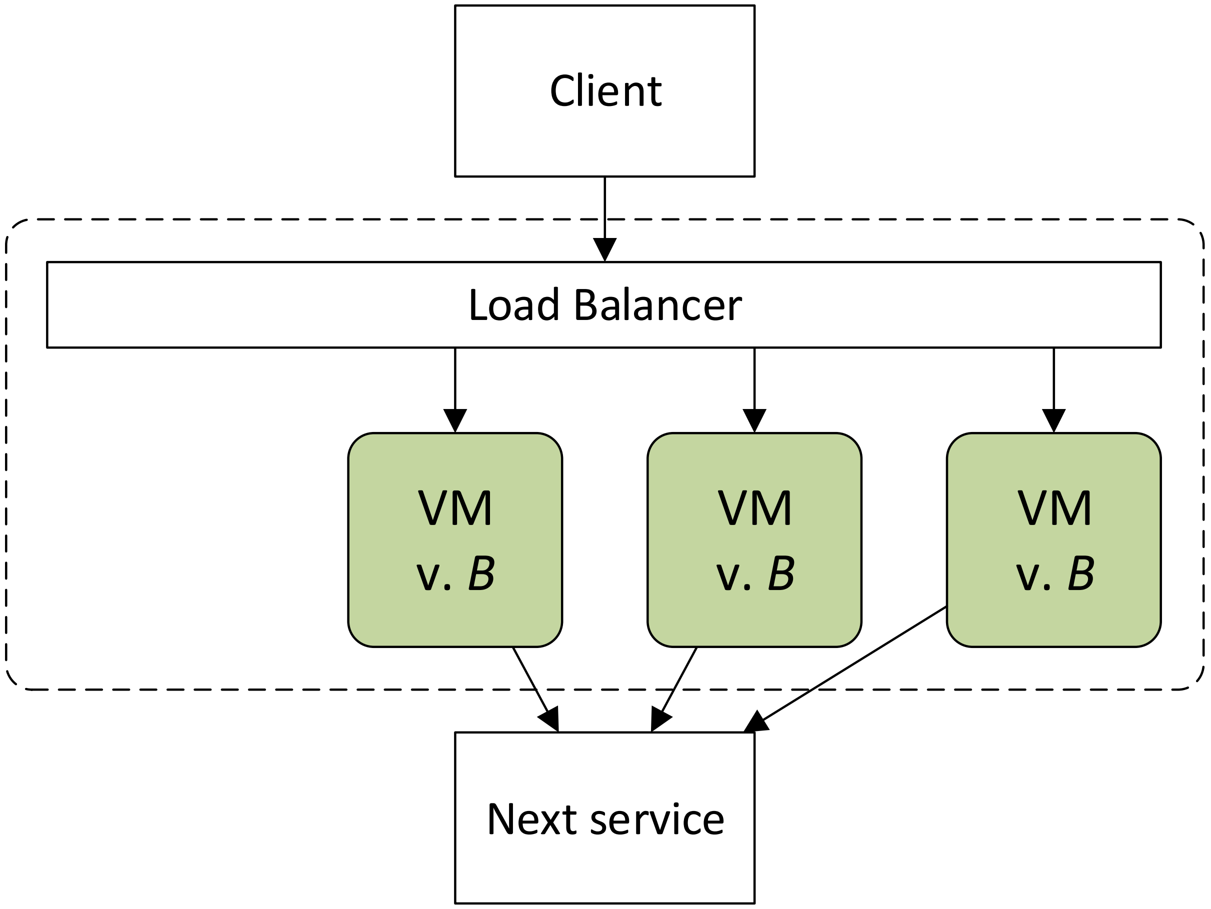
Rolling Upgrade



Rolling Upgrade



Rolling Upgrade



Blue/Green vs. Rolling Upgrade

Blue/Green Deployment

- Only one version available to the client at any time
- Requires 2*N* VMs
  - additional cost
- Rollback is easy

Rolling Upgrade

- Multiple versions are available for service at the same time
- Requires *N+1* VMs
  - Can be done at nearly no extra cost

Canary testing

- Canaries are a small number of instances of a new version placed in production in order to perform live testing in a production environment.
- Canaries are observed closely to determine whether the new version introduces any logical or performance problems. If not, roll out new version globally. If so, roll back canaries.
- Named after canaries

  in coal mines

Implementation of canaries

- Create set of new VMs as canaries (they are unaware of that)
- Designate a collection of customers as testing the canaries. Can be, for example
  - organization-based (dog-fooding)
  - geographically based
  - at random
- Then
  - Route messages from canary customers to canaries
    – Can be done through making registry/load balancer canary aware
  - Observe the canaries closely
  - Decide on rolling out / back

Summary

- Web Services & SOA
  - Service orientation
  - Communication only through standard interfaces
- RESTful services
  - Technology to implement Web services
  - Uniform interface to resources
- DevOps
  - Developers and operators collaborating to joint goal
  - Fast deployment of new features to production while ensuring high quality
- Microservices
  - Split application into small, well-scoped components
- Continuous Deployment
  - Deployment without downtime, e.g. Blue/Green, Rolling Upgrade
  - Live testing

Questions? All makes sense?

Interested in thesis topics, internships, or PhD?
ingo.weber@csiro.au