


DEPAUL UNIVERSITY



# SE480 Week 6 – Modifiability, Other Qualities



Steven Engelhardt



DePaul University

Autumn 2020

# Table of Contents

# Summary of Last Week



- *Performance* is about characterizing the events that can occur (and when they can occur) and the system or element's time-based response to these events.
  - When there is contention for a resource, there is a *scheduling* problem – *something* needs to decide which work gets priority on that scarce resource
  - When investigating performance problems, don't assume you know the cause – let the data and tools guide you
- *Scalability* is about how resource usage changes as units of work grow in number or size.
  - Horizontal scalability is more desirable than vertical scalability, but generally harder to design & implement
  - Using concurrent programming techniques is essential to making programs scale on modern hardware

# Table of Contents

(1 point(s)) Fill in the blank: _____ guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message.

# Question 1



(1 point(s)) Fill in the blank: **Nonrepudiation** guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message.

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure message integrity (with authentication)



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure message integrity (with authentication)



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. **HMAC**
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure confidentiality (shared secret)



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure confidentiality (shared secret)



1. **Symmetric encryption**
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure message integrity (without authentication)



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure message integrity (without authentication)



1. Symmetric encryption
2. Asymmetric encryption
3. **Hashing**
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Protect sensitive data from accidental disclosure



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Protect sensitive data from accidental disclosure



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. **Tokenization**

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure confidentiality (public / private key)



1. Symmetric encryption
2. Asymmetric encryption
3. Hashing
4. HMAC
5. Tokenization

# Question 2



(2 point(s)) Match the following objectives to the most appropriate cryptography technique:

Ensure confidentiality (public / private key)



1. Symmetric encryption
2. **Asymmetric encryption**
3. Hashing
4. HMAC



5. Tokenization

# Question 3



(1 point(s)) You are an architect working on the website for a major bank. You are concerned that a user might connect to the website from a public Wi-Fi access point where network eavesdropping may occur. What cryptographic technique would you use to address this concern?



1. Encryption of data in transit
2. Encryption of data at rest
3. Hashing
4. HMAC

# Question 3



(1 point(s)) You are an architect working on the website for a major bank. You are concerned that a user might connect to the website from a public Wi-Fi access point where network eavesdropping may occur. What cryptographic technique would you use to address this concern?



1. **Encryption of data in transit**
2. Encryption of data at rest
3. Hashing
4. HMAC

# Question 4

(1 point(s)) This question is meant to help illustrate how encryption does not ensure integrity.

It is the year 2100 and the fictional country of Bartovia is at war with the fictional country of Absurdistan. Absurdistan sends all military communiqués using the ROT-13 substitution cipher.

You are a cryptography expert working for Bartovia and you intercept the following encrypted message being sent from Absurdistan's central command to their submarine fleet:

`NGGNPX NG QNJA` (____)

Your anti-submarine warships can intercept Absurdistan's submarines, but they are far away and need time to get there.

You are unable to prevent the message from being delivered entirely, but you *can* change individual characters in the message, but you must keep the overall message length exactly the same. What encrypted message would you send to Absurdistan's fleet to get them to attack at the darker stage of twilight instead of at the first appearance of light in the morning?

# Question 4

(1 point(s)) This question is meant to help illustrate how encryption does not ensure integrity.

It is the year 2100 and the fictional country of Bartovia is at war with the fictional country of Absurdistan. Absurdistan sends all military communiqués using the ROT-13 substitution cipher.

You are a cryptography expert working for Bartovia and you intercept the following encrypted message being sent from Absurdistan's central command to their submarine fleet:

`NGGNPX NG QNJA (ATTACK AT DAWN)`

Your anti-submarine warships can intercept Absurdistan's submarines, but they are far away and need time to get there.

You are unable to prevent the message from being delivered entirely, but you *can* change individual characters in the message, but you must keep the overall message length exactly the same. What encrypted message would you send to Absurdistan's fleet to get them to attack at the darker stage of twilight instead of at the first appearance of light in the morning?

`NGGNPX NG QHFX (ATTACK AT DUSK)`

# Question 5



(1 point(s)) Which security tactic refers to ensuring than an actor has the rights to access and modify either data or services?

1. Identify actors
2. Authenticate actors
3. Authorize actors
4. Limit exposure

# Question 5



(1 point(s)) Which security tactic refers to ensuring than an actor has the rights to access and modify either data or services?



1. Identify actors
2. Authenticate actors
3. **Authorize actors**
4. Limit exposure

# Question 6



(1 point(s)) Which of the below security tactics is *least effective* at detecting attacks?

1. Detect intrusion
2. Detect service denial
3. Verify message integrity
4. Detect message delay

# Question 6



(1 point(s)) Which of the below security tactics is *least effective* at detecting attacks?



1. Detect intrusion
2. Detect service denial
3. Verify message integrity
4. **Detect message delay**

# Question 7

(3 point(s)) According to our lecture, what are three of the 10 most common security vulnerabilities in web applications according to OWASP?

# Question 7

(3 point(s)) According to our lecture, what are three of the 10 most common security vulnerabilities in web applications according to OWASP?



- Injection
- Broken authentication
- Sensitive Data exposure
- XML External Entities (XXE)
- Broken Access Control
- Security Misconfiguration
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- Using Components with Known vulnerabilities
- Insufficient Logging & Monitoring

# Question 8



(1 point(s)) Fill in the blank: _____ is combining a group of components together to verify that they interact correctly together.

# Question 8



(1 point(s)) Fill in the blank: **Integration testing** is combining a group of components together to verify that they interact correctly together

# Question 9



(1 point(s)) Fill in the blank: _____ is evaluating how system behaves under unfavorable conditions

# Question 9



(1 point(s)) Fill in the blank: **Stress testing** is evaluating how system behaves under unfavorable conditions

# Question 10



(1 point(s)) What type of testing is the Apache JMeter tool primarily designed to help with? (You may have to do your own research on this tool)



1. Performance testing
2. Functional testing
3. Integration testing
4. Unit testing

# Question 10



(1 point(s)) What type of testing is the Apache JMeter tool primarily designed to help with? (You may have to do your own research on this tool)



1. **Performance testing**
2. Functional testing
3. Integration testing
4. Unit testing

# Question 11

(1 point(s)) Which of the below is closest to the primary benefit of using a record/playback tool?

1. It allows you to accurately recreate production-like behavior in test environments
2. It allows you to better control the internal state of a component under test
3. It makes the system more convenient to start in an arbitrary state for a test
4. It allows you to substitute test data more easily

# Question 11

(1 point(s)) Which of the below is closest to the primary benefit of using a record/playback tool?

1. **It allows you to accurately recreate production-like behavior in test environments**
2. It allows you to better control the internal state of a component under test
3. It makes the system more convenient to start in an arbitrary state for a test
4. It allows you to substitute test data more easily

# Question 12



(1 point(s)) A developer is asked to fix a defect. They create an automated unit test to find the defect, ensure that the automated unit test fails, fix the defect, and then ensure that the automated unit test succeeds. What testability tactic are they using?



1. Test-driven development
2. Specialized testing interfaces
3. Localize state storage
4. Executable assertions

# Question 12



(1 point(s)) A developer is asked to fix a defect. They create an automated unit test to find the defect, ensure that the automated unit test fails, fix the defect, and then ensure that the automated unit test succeeds. What testability tactic are they using?



1. **Test-driven development**
2. Specialized testing interfaces
3. Localize state storage
4. Executable assertions

# Question 13



(1 point(s)) Fill in the blank: To ____ means to use a control mechanism to coordinate, manage, and sequence the invocation of particular services (which could be ignorant of each other).

(1 point(s)) Fill in the blank: To **orchestrate** means to use a control mechanism to coordinate, manage, and sequence the invocation of particular services (which could be ignorant of each other).

# Question 14

(1 point(s)) Which of the below reasons best describes a reason why you might apply one or more interoperability tactics?

1. You want your system to provide a service to used by unknown systems in the future.
2. You would like your application to consume less battery power on mobile devices.
3. You are ensuring that the system will be available for legitimate use.
4. You are concerned with the ease with which the software that was built to run on one platform can be changed to run on a different platform.

# Question 14

(1 point(s)) Which of the below reasons best describes a reason why you might apply one or more interoperability tactics?

1. **You want your system to provide a service to used by unknown systems in the future.**
2. You would like your application to consume less battery power on mobile devices.
3. You are ensuring that the system will be available for legitimate use.
4. You are concerned with the ease with which the software that was built to run on one platform can be changed to run on a different platform.

# Question 15



(1 point(s)) You are designing an application using the microservices pattern and trying to decide the best way for one internal service to connect to another within your data center. If the connection from one service to another is *performance-critical*, what is the most appropriate interoperability tactic to use?



1. Client-side discovery
2. Server-side discovery
3. Service registry
4. Orchestrate

# Question 15



(1 point(s)) You are designing an application using the microservices pattern and trying to decide the best way for one internal service to connect to another within your data center. If the connection from one service to another is *performance-critical*, what is the most appropriate interoperability tactic to use?



1. **Client-side discovery**
2. Server-side discovery
3. Service registry
4. Orchestrate

# Question 16



(1 point(s)) Postel's Law states "Be conservative in what you send, be liberal in what you accept." What is the most likely reason why some architects disagree with following this "law"?



1. It reduces the interoperability of an implementation
2. It reduces the performance of an implementation
3. It causes deviations to become entrenched and increases long-term costs
4. It leads to unavoidable security flaws in an implementation

# Question 16



(1 point(s)) Postels Law states "Be conservative in what you send, be liberal in what you accept." What is the most likely reason why some architects disagree with following this "law"?



1. It reduces the interoperability of an implementation
2. It reduces the performance of an implementation
3. **It causes deviations to become entrenched and increases long-term costs**
4. It leads to unavoidable security flaws in an implementation

# Question 17



(1 point(s)) True or false: Standards, by themselves, are enough to guarantee interoperability.



1. False
2. True

# Question 17



(1 point(s)) True or false: Standards, by themselves, are enough to guarantee interoperability.



1. **False**
2. True

# Table of Contents

# HW2 Review

Write a short essay (800-1,000 words) on the topic of "The Role of the Architect". Based on the article, as well as what we've discussed in class and your own work/life experiences, summarize the role of the architect and its importance in software engineering.

Here are some questions you should consider while writing your essay:

- Do you agree with Dr. Krutchen's assessment of the role of the architect? Why or why not?
- Do you agree with the point of view presented in your textbook or by me? Why or why not?
- In your opinion, what separates a good architect from a bad architect?
- Do you believe that dedicated architects are necessary for the success of an organization?
- If you were the Head of Software Architecture at your organization, how would you measure and evaluate your architecture team's success?

# Table of Contents

# Table of Contents

# What is Modifiability?



- *Modifiability* is about managing the cost and risk of making changes to the system
- Four basic questions:
  1. What can change?
  2. What is the likelihood of the change?
  3. When is the change made and who makes it?
  4. What is the cost of the change?

# Why is Modifiability Important?



- Most of the cost of the typical software system occurs after it has been initially released
- Software change is not only constant but ubiquitous

# Modification – What Can Change?



- Functions
- Platform
- Environment
- Qualities
- Capacity

# Modification – What is Likelihood of the Change?



- Architect has to make tough decisions (a.k.a. educated guesses) about which changes are likely and hence which changes are to be supported, and which are not.

# Modification – When and By Whom?

- *Implementation time* (programmers modifying source code)

# Modification – When and By Whom?

- *Implementation time* (programmers modifying source code)
- *Compile time* (build managers using compile-time switches)

# Modification – When and By Whom?

- *Implementation time* (programmers modifying source code)
- *Compile time* (build managers using compile-time switches)
- *Build time* (release managers by choosing different libraries)

# Modification – When and By Whom?

- *Implementation time* (programmers modifying source code)
- *Compile time* (build managers using compile-time switches)
- *Build time* (release managers by choosing different libraries)
- *Configuration setup time* (deployment managers by setting different config options)

# Modification – When and By Whom?

- *Implementation time* (programmers modifying source code)
- *Compile time* (build managers using compile-time switches)
- *Build time* (release managers by choosing different libraries)
- *Configuration setup time* (deployment managers by setting different config options)
- *Execution time* (operators by changing parameter settings or using plugins)

# Modification – When and By Whom?



- *Implementation time* (programmers modifying source code)
- *Compile time* (build managers using compile-time switches)
- *Build time* (release managers by choosing different libraries)
- *Configuration setup time* (deployment managers by setting different config options)



- *Execution time* (operators by changing parameter settings or using plugins)
- *Binding time* refers to the stage in the life cycle that the modification takes place.
  - On average, a architecture that is suitably equipped to accommodate modifications late in the life cycle will, on average, cost less than an architecture that forces the same modification to be made earlier.

# Modification – What is the Cost of the Change?

- The *cost of changes* includes:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)

# Modification – What is the Cost of the Change?

- The *cost of changes* includes:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)
- For $N$ similar modifications, the *justification* is:

$$N \times \text{Cost of making the change without the mechanism} \leq$$
$$\text{Cost of installing the mechanism} +$$
$$(N \times \text{Cost of making the change using the mechanism})$$

# Modification – What is the Cost of the Change?

- The *cost of changes* includes:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)
- For $N$ similar modifications, the *justification* is:

$$N \times \text{Cost of making the change without the mechanism} \leq \text{Cost of installing the mechanism} + (N \times \text{Cost of making the change using the mechanism})$$

- We may not need to completely change mechanisms to reduce change costs!
  - What if we invested in tools, processes, etc. to make source code changes less expensive?

# Coupling

- *Coupling* is a measure of the strength of the association established by a connection from one module to another.
- Your shirt has *loose coupling* to your body. Your skin as *tight coupling* to your body.
- Tight coupling is often called by two modules having *overlapping responsibilities*
- Disadvantages of tight coupling:
  - A change in one module usually forces a ripple effect of changes in other modules.
  - Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
  - A particular module might be harder to reuse and/or test because dependent modules must be included.

# Cohesion



- *Cohesion* measures how strongly the responsibilities of a module are related. Informally, it measures the module's "unity of purpose".
- Advantages of *strong cohesion* are:
  - Reduced module complexity (they are simpler, having fewer operations).
  - Increased system maintainability, because logical changes in the domain affect fewer modules, and because changes in one module require fewer changes in other modules.
  - Increased module reusability, because application developers will find the component they need more easily among the cohesive set of operations provided by the module.

# Modifiability Summary



- All else equal, we want systems with:
  - Fewer lines of code
  - Loose coupling
  - Strong cohesion
  - Late binding time

# Table of Contents

# Modifiability Tactic Categories



- Reduce size of a module
- Increase cohesion
- Reduce coupling
- Defer binding

# Split Module

- Separate a module into several smaller modules
- Common criterion is if children can be modified independently
- Splitting can support deferred bindings
  - Partition 1: Manages primary activity
  - Partition 2: Manages runtime activities to support a request for modification
- What tactic does this remind you of?

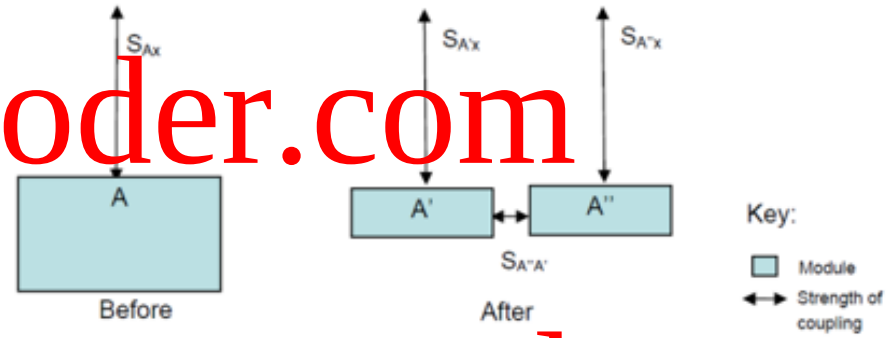



Figure: Split Module

# Split Module

- Separate a module into several smaller modules
- Common criterion is if children can be modified independently
- Splitting can support deferred bindings
  - Partition 1: Manages primary activity
  - Partition 2: Manages runtime activities to support a request for modification
- What tactic does this remind you of?
  - Non-stop forwarding



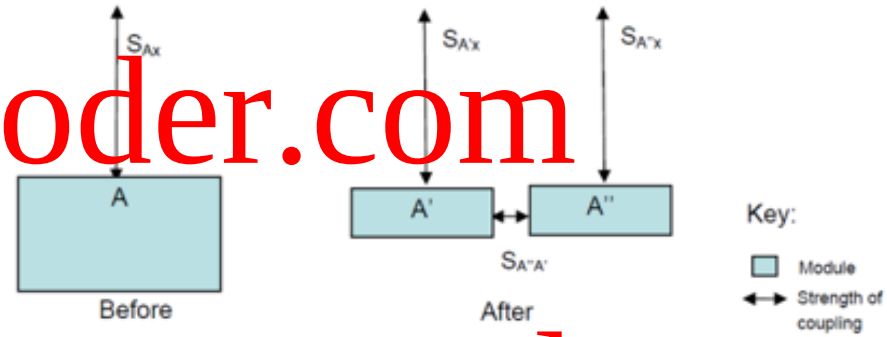



Figure: Split Module

# Increase Semantic Coherence



- If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules.
- This may involve creating a new module or it may involve moving a responsibility to an existing module.

# Encapsulate

- Introduce an explicit interface to a module to limit the ways in which external responsibilities can interact with the module
- Interface may have its own responsibilities such as "transform input parameter to internal representation"
- One of Java 9's major features was a *Platform Module System* which allows stronger encapsulation of Java code

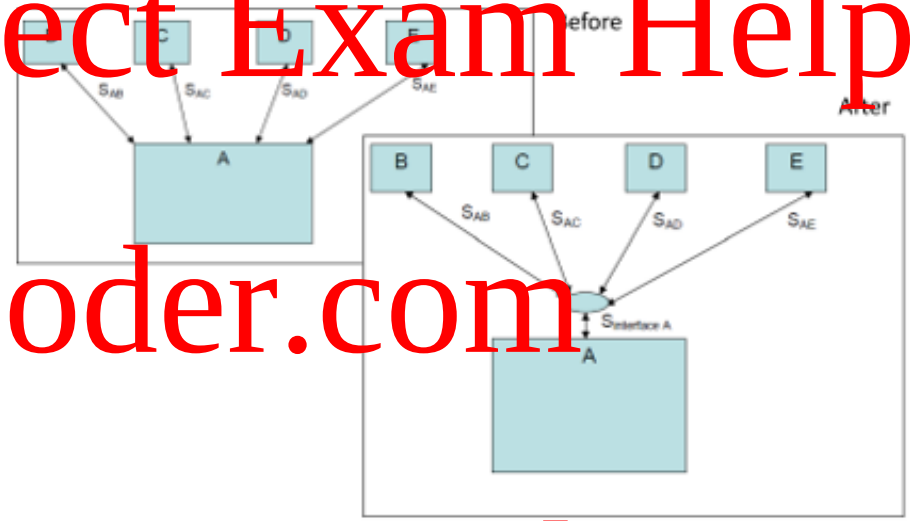



Figure: Encapsulate

*Asymmetrical coupling*: $A \rightarrow I(A)$ strong, $I(A) \rightarrow A$ weak

# Use an Intermediary



- An *intermediary* breaks a dependency
- Example: If A is a producer and B is a consumer, then a publish-subscribe intermediary removes A's knowledge of B. It can also remove partial knowledge of B for A.
- A SOA *directory service* is an intermediary





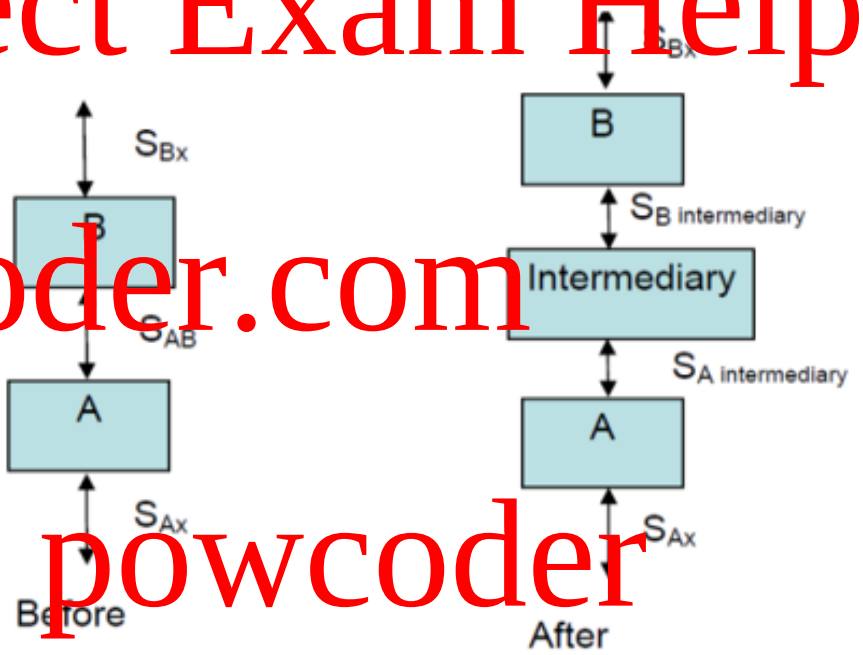



Figure: Use an Intermediary

# Restrict Dependencies



- Restricts the modules that a given module interacts with or depends upon.
- Achieved by restricting a module's visibility (when developers cannot see an interface, they cannot employ it) and by authorization.
- Many times, *fewer dependencies* implies *more reusability*
  - A potential client of your module must evaluate introducing not only you but also all of your dependencies

# Abstract Common Services



- If A and B represent similar services, abstract out A and B and implement once in a slightly more general form
- In *refactoring*, two modules can be joined and re-split to increase coherence



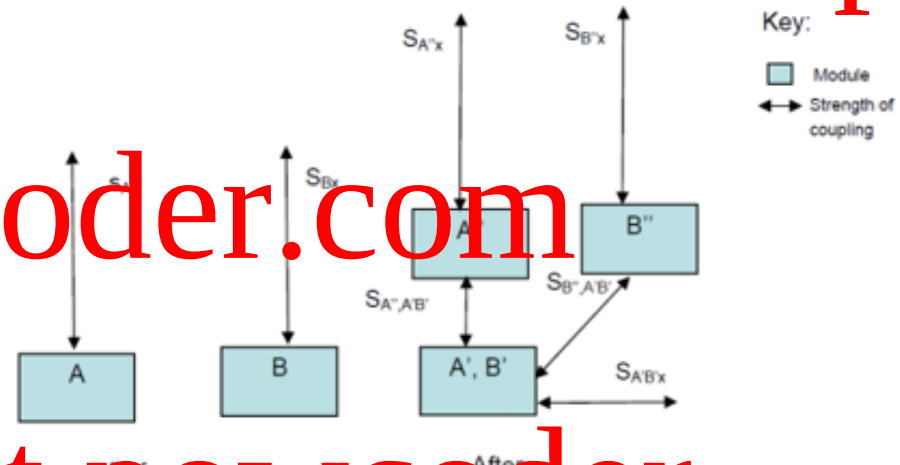





Figure: Abstract Common Services

# Defer Binding

- Goal: Move binding later in the lifecycle
- *Compile-time binding*
  - Component replacement
  - Compile-time parameter parameterization
  - Aspects
- *Deployment-time binding*
  - Configuration-time binding
- *Startup / Initialization-time binding*
  - Resource files
- *Runtime binding*
  - Runtime registration
  - Dynamic lookup (e.g. for services)
  - Interpret parameters
  - Startup time binding
  - Name servers
  - Plug-ins
  - Publish-subscribe
  - Shared repositories
  - Polymorphism
- *Externalize the change*: Install a mechanism so that someone else can make a change to the system without having to change any code

# Table of Contents

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
- ***O**pen/closed principle*: Software entities should be open for extension, but closed for modification.

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
- ***O**pen/closed principle*: Software entities should be open for extension, but closed for modification.
- ***L**iskov substitution principle*: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
- ***O**pen/closed principle*: Software entities should be open for extension, but closed for modification.
- ***L**iskov substitution principle*: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- ***I**nterface segregation principle*: Many client-specific interfaces are better than one general-purpose interface.

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
- ***O**pen/closed principle*: Software entities should be open for extension, but closed for modification.
- ***L**iskov substitution principle*: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- ***I**nterface segregation principle*: Many client-specific interfaces are better than one general-purpose interface.
- ***D**ependency inversion principle*: One should depend upon abstractions, not concretions.

# SOLID (Object-Oriented Design)

- ***S**ingle responsibility principle*: A class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
- ***O**pen/closed principle*: Software entities should be open for extension, but closed for modification.
- ***L**iskov substitution principle*: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- ***I**nterface segregation principle*: Many client-specific interfaces are better than one general-purpose interface.
- ***D**ependency inversion principle*: One should depend upon abstractions, not concretions.
- How do these principles relate to software architecture?

# Inversion of Control



- *Inversion of control* is a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework
- Increases modularity of a program and makes it extensible
- Techniques:
  - Factory design pattern
  - Service locator
  - Dependency injection
  - Strategy design pattern

# Factory Pattern

- A *factory* is an object for creating other objects
- Use when:
  - The creation of an object makes reuse impossible without significant duplication of code.
  - The creation of an object requires access to information or resources that should not be contained within the composing class.
  - The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.



```
public class ImageReaderFactory {
  public static ImageReader createImageReader(
        ImageInputStreamProcessor iisp) {
    if (iisp.isGIF()) {
      return new GifReader(iisp.getInputStream());
    } else if (iisp.isJPEG()) {
      return new JpegReader(iisp.getInputStream());
    } else {
      throw new
        IllegalArgumentException("Unknown image type.");
    }
  }
}
```

# Dependency Injection



- A technique whereby one object supplies the dependencies of another object
  - The service is passed to the client, rather than the client trying to build or find the service
- Solves the following problems:
  - How can an application be independent of how its objects are created?
  - How can a class be independent of how the objects it requires are created?
  - How can the way objects are created be specified in separate configuration files?
  - How can an application support different configurations?
- Earlier this lecture, we also saw how dependency injection helps with testability

# Dependency Injection Example

```
public class FinalInvoiceStep {
  private PrinterServiceImpl printerService = null;
  private EmailServiceImpl emailService = null;

  public FinalInvoiceStep() {
    this.printerService = new PrinterServiceImpl();
    this.emailService = new EmailServiceImpl();
  }

  public void handleInvoice(Invoice invoice,
      Customer customer) {
    if (customer.prefersEmails()) {
      emailService.sendInvoice(invoice,
          customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public void testFinalInvoiceStep() {
  FinalInvoiceStep finalInvoiceStep =
      new FinalInvoiceStep();
  // ???
}
```

Listing 1: Without Dependency Injection

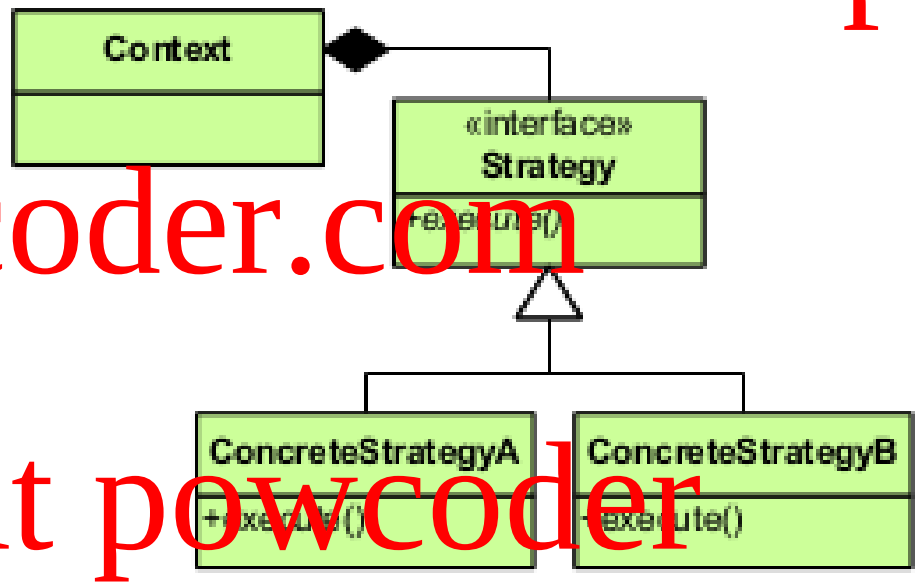
```
public class FinalInvoiceStep {
  private IPrinterService printerService = null;
  private IEmailService emailService = null;

  public FinalInvoiceStep(
      IPrinterService printerService,
      IEmailService emailService) {
    this.printerService = printerService;
    this.emailService = emailService;
  }

  public void handleInvoice(Invoice invoice,
      Customer customer) {
    if (customer.prefersEmails()) {
      emailService.sendInvoice(invoice,
          customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}
```

Listing 2: With Dependency Injection

# Strategy Pattern



- A *strategy* enables selecting an algorithm at runtime
- Examples of when to use:
  - A resource allocator. Each strategy here has an "Allocate" method that has the same interface, with the user making a decision about which strategy to use based on what they are trying to optimize.
  - When you want to use a different algorithm for "large" data sets vs "small" data sets (e.g. sorting)



# Strategy Pattern Example



```
// Strategy Interface
public interface CompressionStrategy {
  public void compressFiles(ArrayList<File> files);
}

public class ZipCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    // using ZIP approach
  }
}

public class SnappyCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    // using Snappy approach
  }
}
```

# Aspect-Oriented Programming



- A *cross-cutting concern* is an aspect of a program that affect other concerns, and often cannot be cleanly decomposed from the rest of the system
- *Aspect-oriented programming* is about separating cross-cutting concerns
- Examples
  - Logging
  - Authorization

# Aspect-Oriented Programming Example

```
void transfer(Account fromAcc, Account toAcc,
    int amount, User user, Logger logger,
    Database database) throws Exception {
  logger.info("Transferring money...");

  if (!isUserAuthorised(user, fromAcc)) {
    logger.info("User has no permission.");
    throw new UnauthorisedUserException();
  }

  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);

  database.commitChanges(); // Atomic operation.

  logger.info("Transaction successful.");
}
```

Listing 3: Without aspect-oriented programming

```
void transfer(Account fromAcc, Account toAcc,
    int amount) throws Exception {
  if (!isUserAuthorised(user, fromAcc))
    throw new UnauthorisedUserException();
  if (fromAcc.getBalance() < amount)
    throw new InsufficientFundsException();

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);
  database.commitChanges();
}

aspect Logger {
  void Bank.transfer(Account fromAcc,
      Account toAcc, int amount,
      User user, Logger logger)  {
    logger.info("Transferring money...");
  }

  void Bank.getMoneyBack(User user,
      int transactionId, Logger logger)  {
    logger.info("User requested money back.");
  }
}
```

Listing 4: With aspect-oriented programming

# Disadvantages of Aspect-Oriented Programming



- Obscures control flow
- Extremely sensitive to changes in the program
- Developers seem to be adopting *attribute-oriented programming* instead

# Attribute-Oriented Programming



- Programmers mark program elements (e.g. classes and methods) to indicate that they maintain application-specific or domain-specific semantics
- Either a compile-time tool (e.g. a preprocessor) or a runtime tool is responsible for interpreting these attributes and acting upon them
- You can define your own custom attributes and build your own tools to interpret and enforce them
- Unlike AOP or configuration files, keeps the cross-cutting concerns close to the control flow

# Attribute-Oriented Programming Example 1



```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {
  @Test
  public void multiplicationOfZeroIntegersShouldReturnZero() {
    MyClass tester = new MyClass();

    assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
    assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
    assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
  }
}
```

Listing 5: JUnit 5

# Attribute-Oriented Programming Example 2



```
@WebService
public class BusinessService
{
  @WebMethod
  public string myBusinessMethod() {
    .
    .
    .
  }
}
```

Listing 6: Web Services

# Attribute-Oriented Programming Example 3



```
public interface BankService {
  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
}
```

Listing 7: Spring Security

# Table of Contents

# Classifying Software Changes



- *Adaptive changes*: add new functionality to a system. For example:
  - Caller ID in a telephone switch
  - Adapt software to new hardware or environmental changes
- *Corrective changes*: fix faults in the software
- *Perfective changes*: improve the developers' ability to maintain software

# What is Code Decay?



- A unit of code (in most cases, a module) is *decayed* if it is harder to change that it should be, measured in terms of effort, interval, and quality.
- How to measure code decay:
  - *Cost* of the change, i.e. personnel cost for making the change
  - *Interval* to complete the change (i.e. calendar time)
  - *Quality* of the changed software
- Evidence that code does decay:
  - The *span of changes*, which is shown to increase over time
  - *Breakdown of modularity*, which is exhibited by means of network style visualizations
  - *Fault potential*, which is the likelihood of changes to induce faults in the software system

# Causes of Code Decay

- *Inappropriate architecture* that does not support the changes or abstractions required of the system
- *Violations of the original design principles*
  - Changes that match original design tend to be comparatively easy
  - For example, in switching systems, many of the original system abstractions assume that subscriber phones remain in fixed locations. Changes needed to support wireless phones that roam among cell sites were unanticipated by original system designers
- *Imprecise requirements* introduce the need for frequent corrective changes
- *Time pressure* leads programmers to take shortcuts (i.e. incur technical debt)
- *Inadequate programming tools* so that refactoring is not supported
- *Organizational environment* manifested in low morale, excessive turnover, or inadequate communication among stakeholders
- *Programmer variability*, i.e. programmers who cannot or change delicate, complex code written by their more skilled colleagues
- *Inadequate change processes* such as lack of a version control system or inability to handle parallel changes

# Symptoms of Code Decay



- *Excessively complex code*
- *A history of frequent changes* suggests prior repair and modifications
  - Also referred to as *code churn*
- A *history of faults*, which is a strong predicter of future faults
- *Widely dispersed changes*
  - Well-engineered, modularized code should lead to local changes
- *Kludges*, which are changes that could have been done better
- *Numerous interfaces*, which leads to increasing attention needed to be directed at possible side effects in other areas of code

- Required Reading: S. Eick, T. Graves, A. Karr, J.S. Marron, and A. Mockus "Does Code Decay? Assessing the Evidence From Change Management Data" *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, January 2001.

# Table of Contents

# Deployability



- *Deployability* is concerned with how an executable arrives at a host platform and how it is subsequently invoked
- Common techniques include
  - Deployment automation tools
  - Self-updating software
  - Managing the associated risk of updates

# Continuous Delivery

- *Continuous integration* is the practice of merging all developers' working copies to a shared mainline several times a day. [Wik20c]
- *Continuous delivery* is a software engineering approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time and, when releasing the software, doing so manually. [Wik20a]
- *Continuous deployment* is a software engineering approach in which software functionalities are delivered frequently through automated deployments. CD contrasts with continuous delivery, a similar approach in which software functionalities are also frequently delivered and deemed to be potentially capable of being deployed but are actually not deployed. [Wik20b]

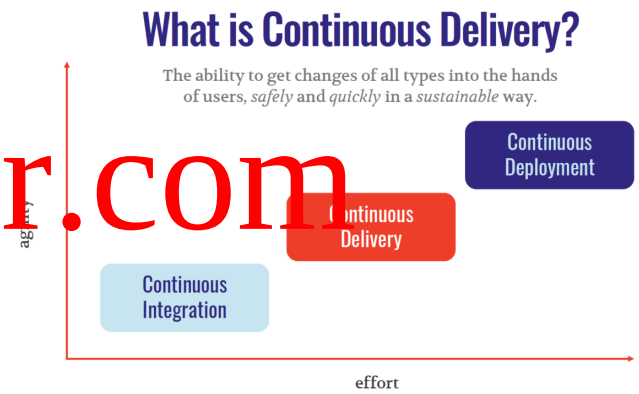



Figure: What is Continuous Delivery? [Shr20]

# Code Pipelines



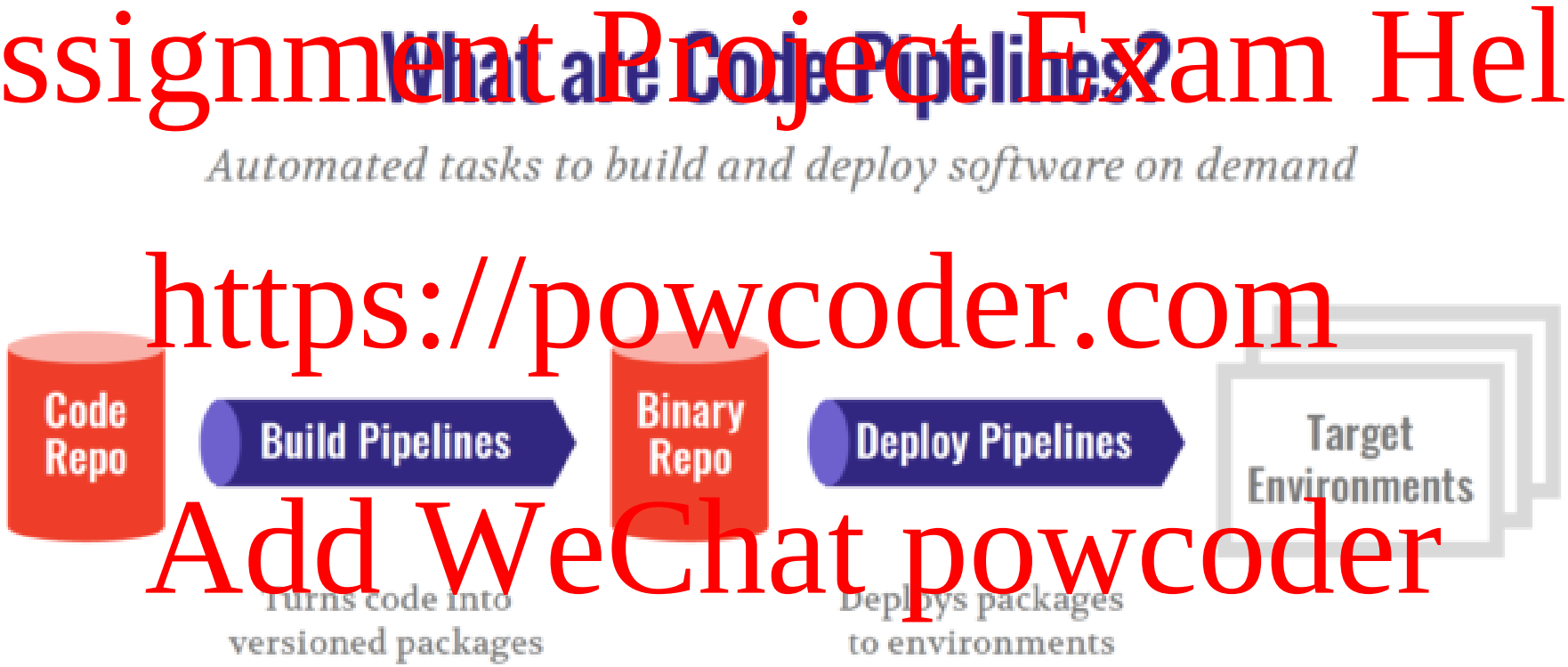





Figure: Relationship of Build and Deploy [Shr20]

# Example Continuous Deployment Pipeline: Overall



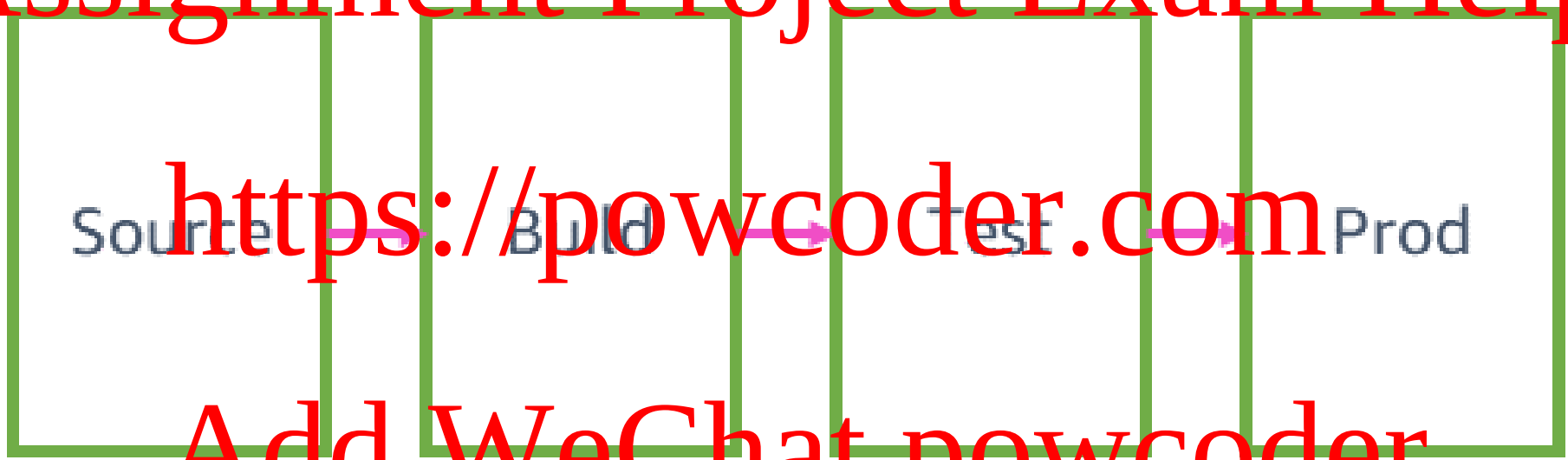



Figure: Example Continuous Deployment Pipeline: Overall [Lig20]

# Example Continuous Deployment Pipeline: Source and Build



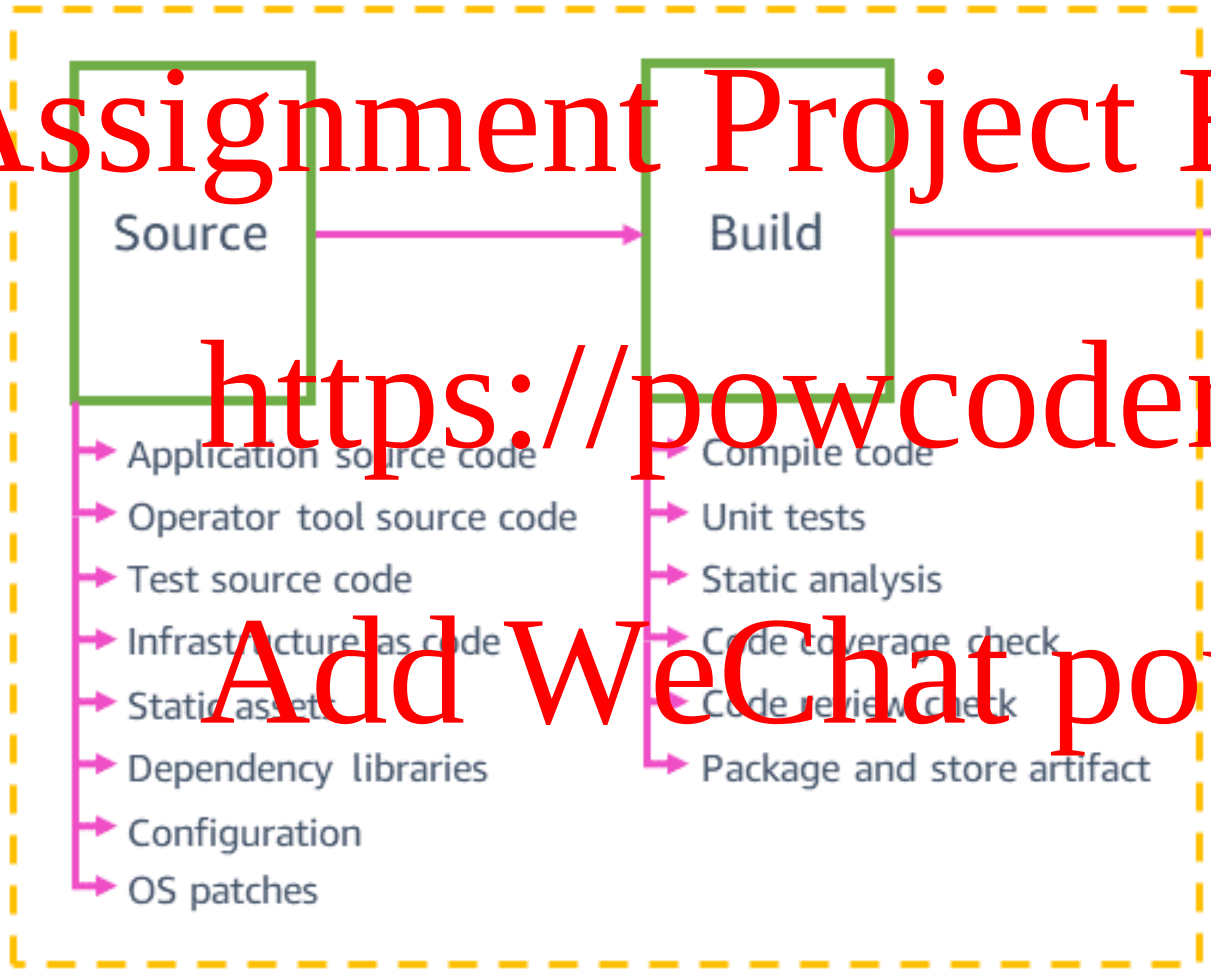



Figure: Example Continuous Deployment Pipeline: Source and Build Stages [Lig20]

# Example Continuous Deployment Pipeline: Test



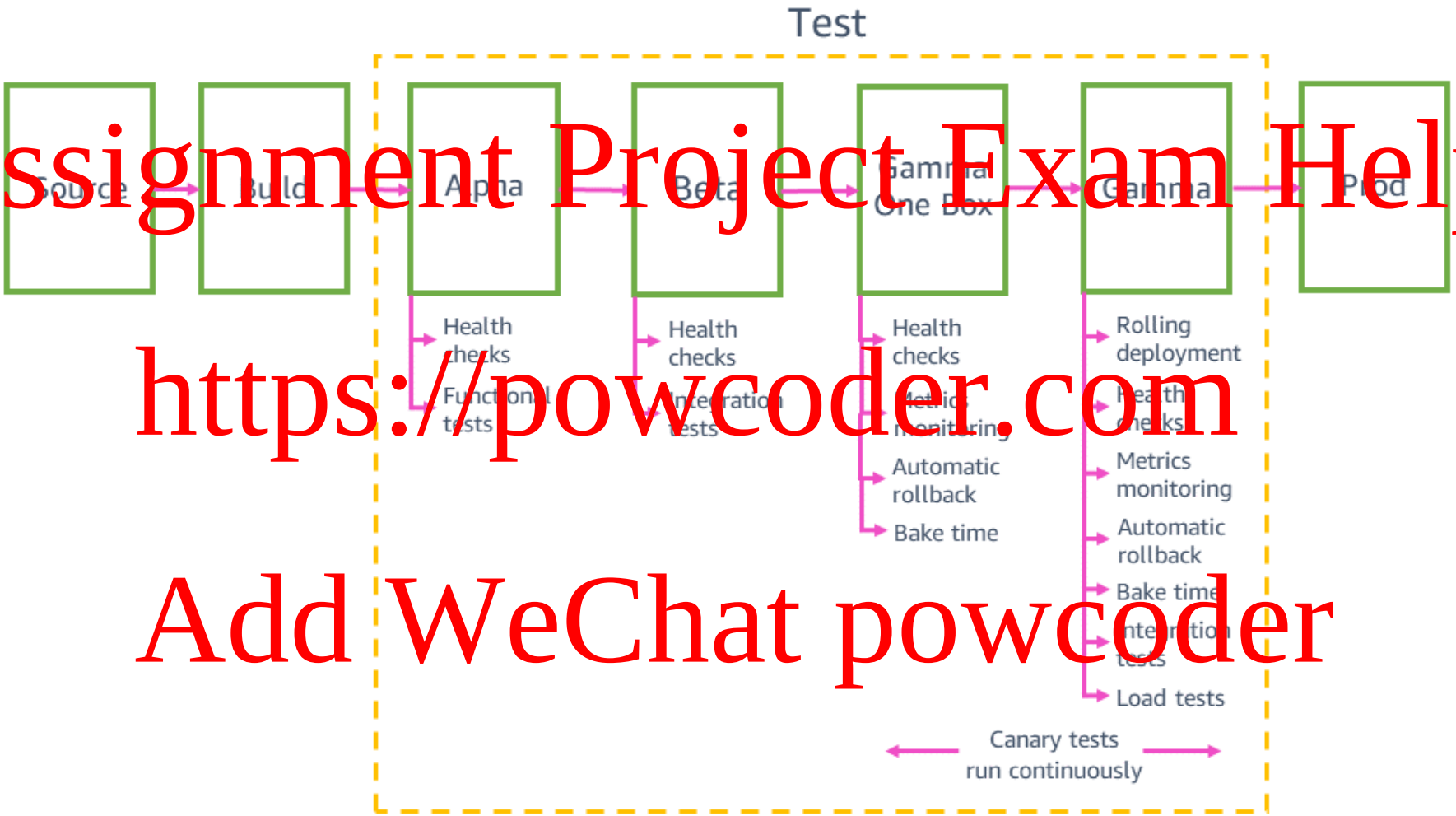



Figure: Example Continuous Deployment Pipeline: Test Stages [Lig20]

# Example Continuous Deployment Pipeline: Prod



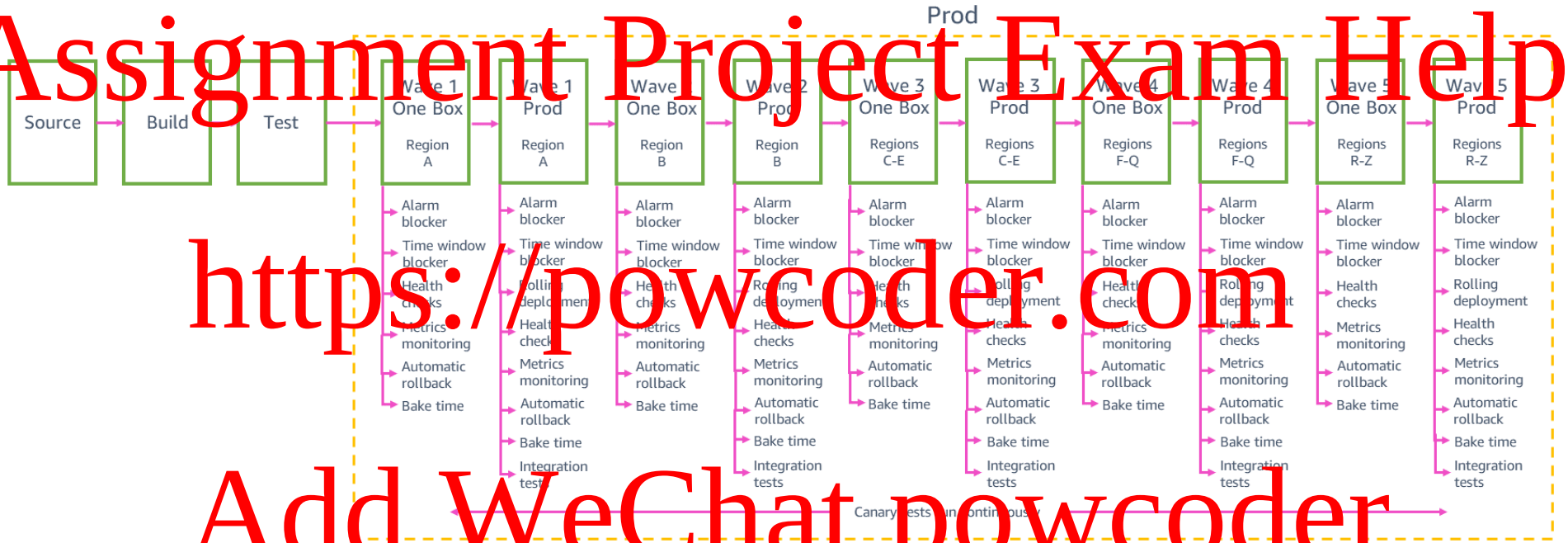



Figure: Example Continuous Deployment Pipeline: Prod Stages [Lig20]

# Ring-Based Deployments

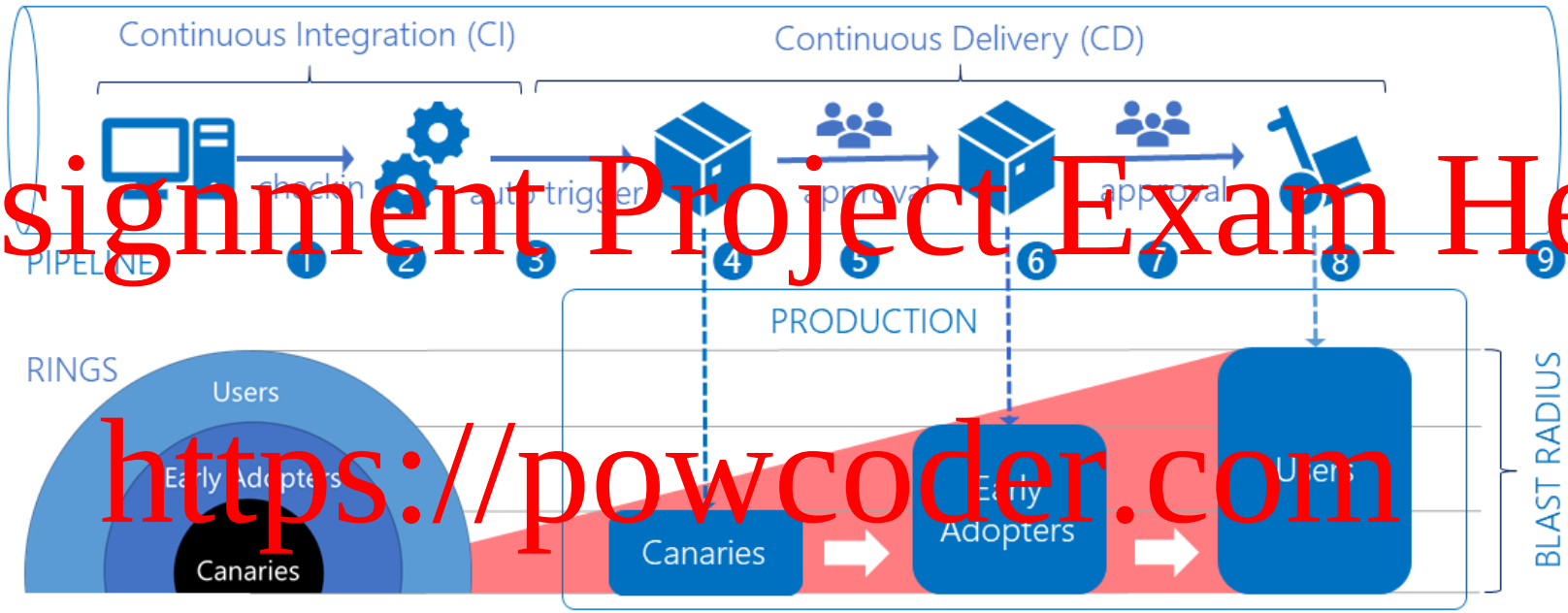





Figure: Deployment Rings [GS18]

- *Deployment rings* (or *deployment waves*) upport the production-first DevOps mindset and limit impact on end users, while gradually deploying and validating changes in production.
- Impact (also called *blast radius*), is evaluated through observation, testing, analysis of telemetry, and user feedback.

# Feature Flags



- *Feature flags* allow you to deploy new features into a production environment, but restrict their availability. With the flick of a switch, you can activate a new feature for specific users without restarting the app or deploying new code.






Figure: Feature Flags

- Feature flags separate the release of new features from their code deployment.

# Rollback Safety

- Amazon engineering guiding tenant: *Avoid walking through one-way doors* [Pok20]
- One way to ensure that you can safely roll back is by using a technique commonly referred to as *two-phase deployment*



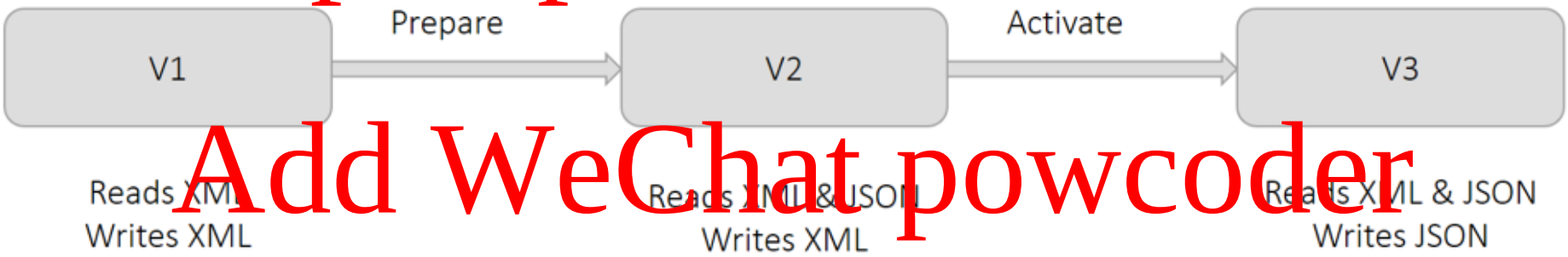



Figure: Two Phase Deployment

# Immutable Infrastructure

- *Immutable infrastructure* refers to servers (or VMs) that are never modified after deployment
- When you do need to update your server, you'll replace it with a new version. For any updates, fixes, or modifications, you'll:
  - Build a new server from a common image, with appropriate changes, packages, and services included
  - Provision the new server to replace the old one
  - Validate the server
  - Decommission the old server
- Advantages of immutable infrastructure include:
  - Known-good server state and fewer deployment failures
  - No configuration drift or snowflake servers
  - Consistent staging environments and easy horizontal scaling
  - Simple rollback and recovery processes

# Optional Readings

- Rebuilding Linkerd's continuous integration (CI) with Kubernetes in Docker (kind) and GitHub Actions
- How we upgraded PostgreSQL at GitLab.com
- Managing feature toggles in teams
- A Deep Dive into How .NET Builds and Ships
- Automating safe, hands-off deployments
- 7 Pipeline Design Patterns for Continuous Delivery
- Deploy your pull requests with GitHub Actions and GitHub Deployments
- Deploys at Slack
- A Practical Guide to Deploying Microservices on AWS
- Deploy on Fridays, or Don't.

# Table of Contents

# Development Distributability



- *Development distributability* is the quality of designing the software to support distributed software development
- The key technique here is to *minimize coordination* among teams
- Remind you of the scalability discussion?

# Component vs. Feature Teams

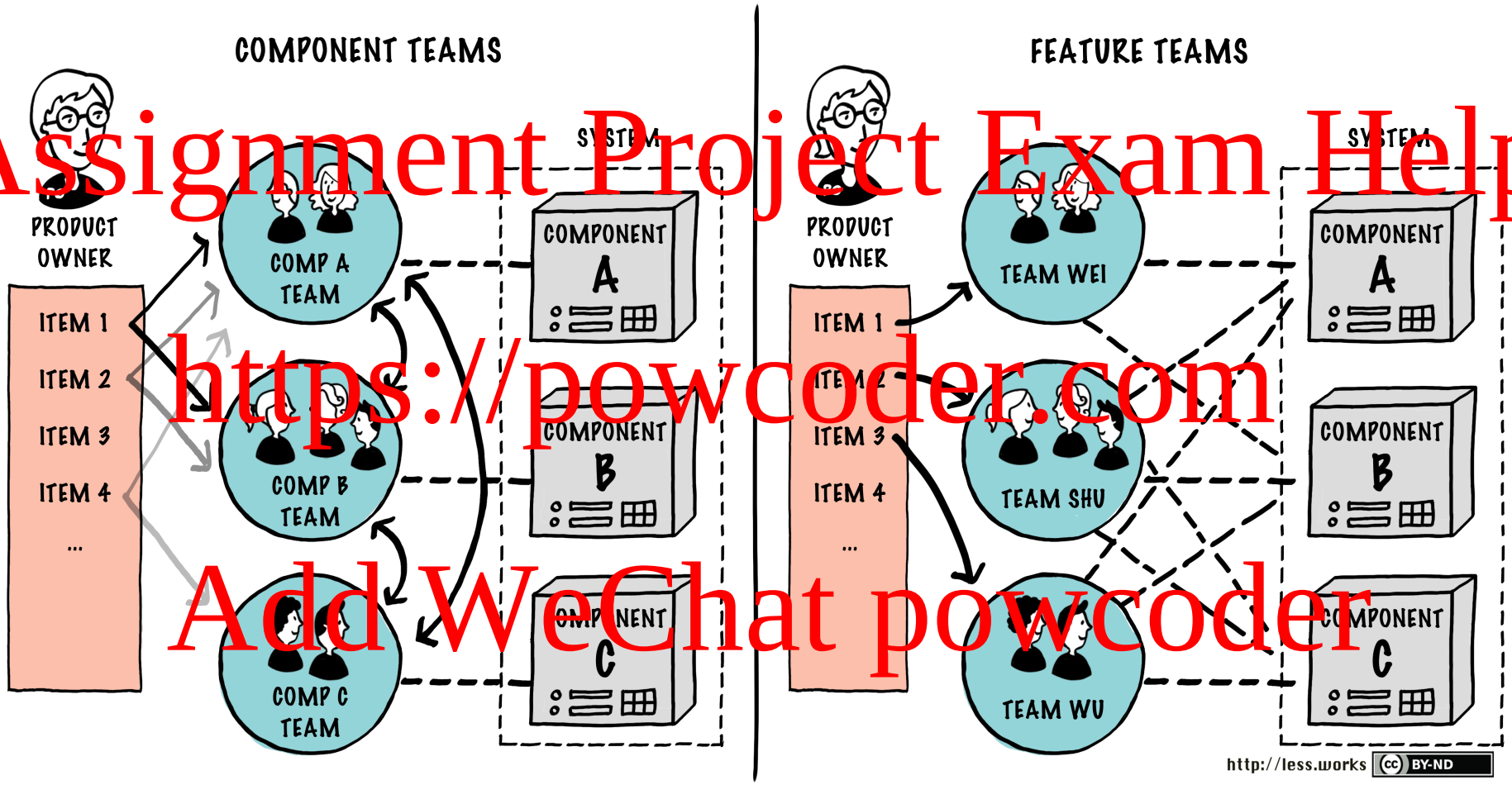





Figure: Component vs. Feature Teams [LeS19]

# Table of Contents

# Mobility



- *Mobility* deals with the problems of movement and affordances of a platform
- Common issues include:
  - Battery management
  - Operating in a disconnected mode
  - Managing multiple user interfaces across multiple platforms
- How many mobile phone apps have the same interface with their desktop and iPad equivalents?

# Example Mobile App Architecture

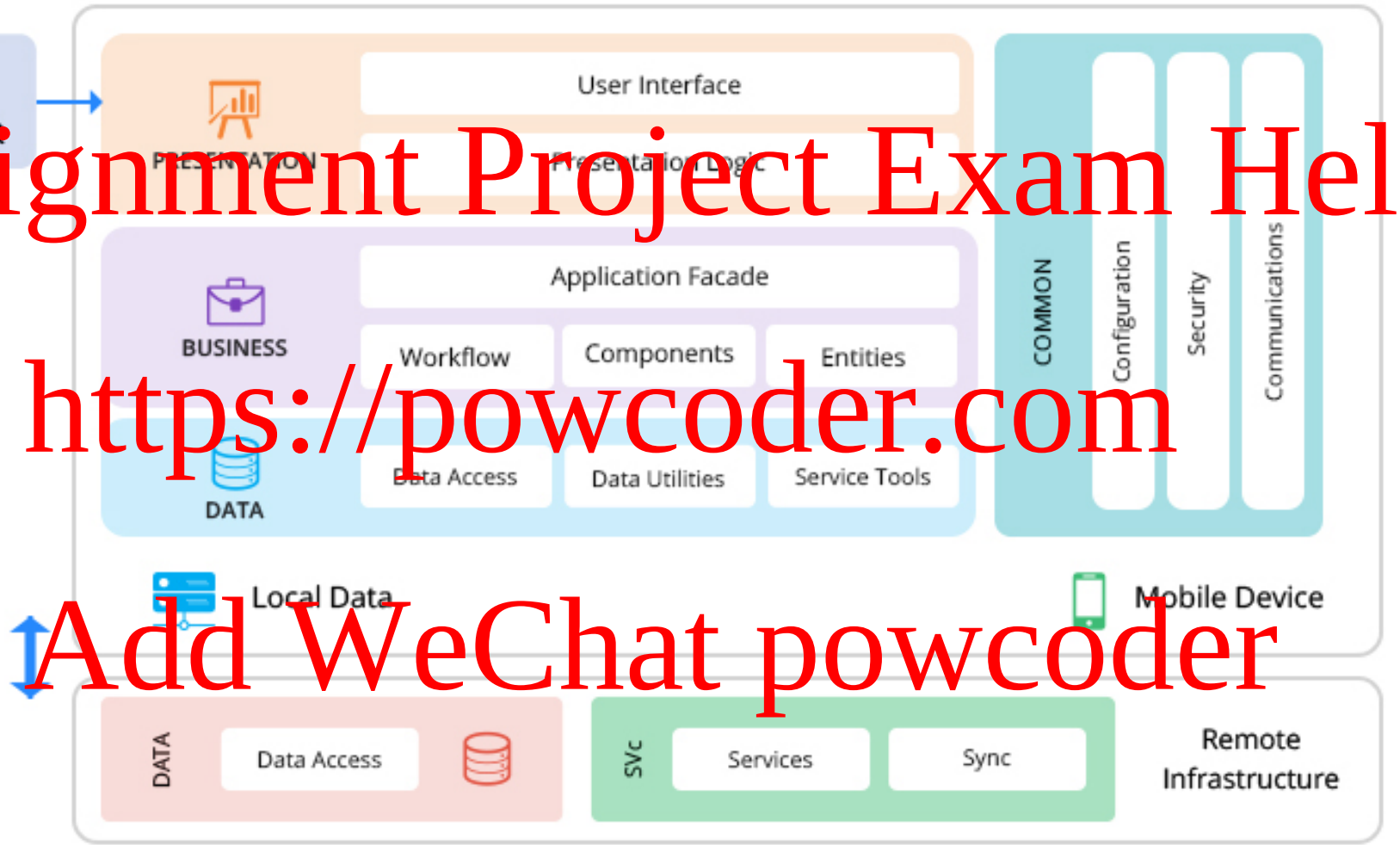





Figure: Example Mobile App Architecture [Bes18]

# Table of Contents

# Observability



- *Observability* (a.k.a. *monitorability*) refers to the ability of the operations staff to monitor the system while it is executing
- Alternatively, observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs. If a system is not observable, this means that the current values of some of its state variables cannot be determined through output sensors.
- Building *telemetry* into your system early often pays off immensely, as it helps provide the data to diagnose performance issues

# Observability Data Types



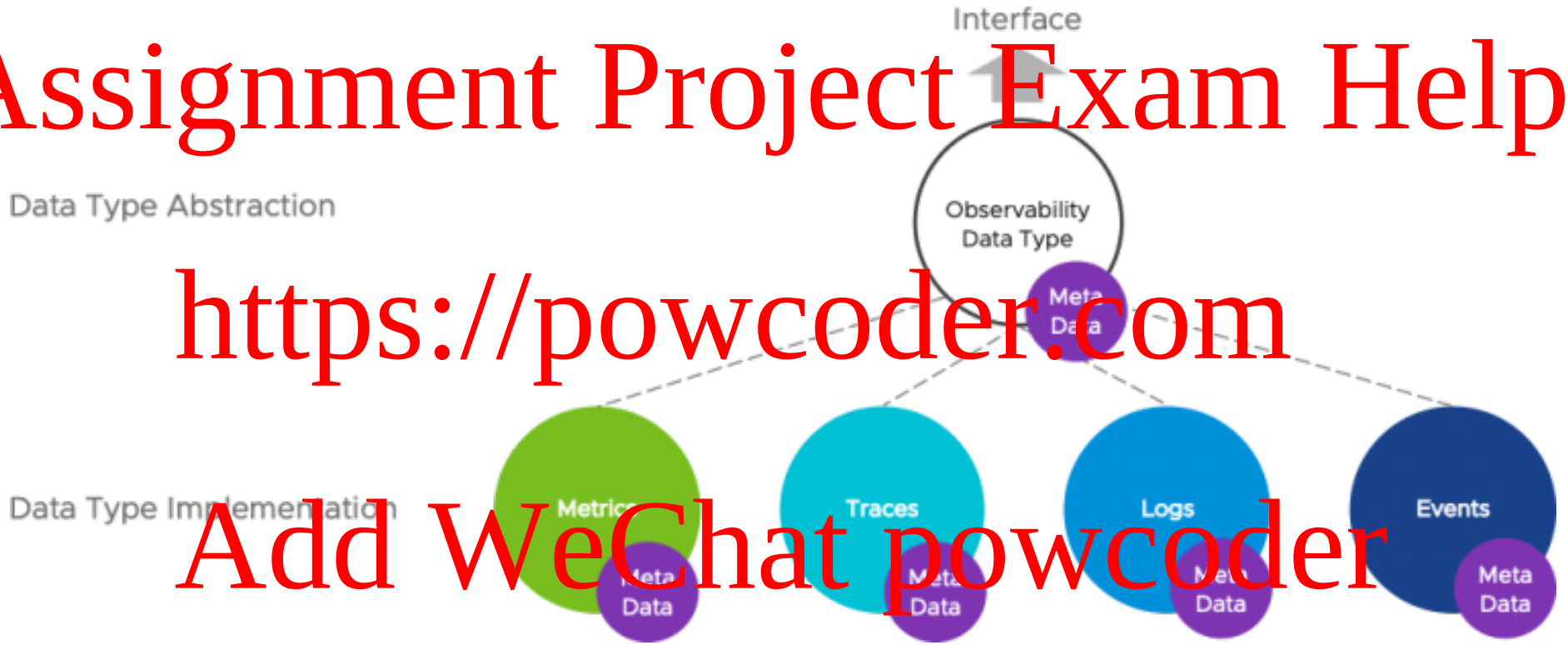



Figure: Observability Data Types

# Event Logs

- An *event log* is an immutable, timestamped record of discrete events that happened over time.
- Logs are often aggregated into a *log aggregator* like Splunk or DataDog, which enables querying, dashboarding, etc.



```
Apr 29 19:13:01 45.2.98.7 SentriantGenericAlert: Time="04/29/06 07:12 PM PDT",Host="roach_motel.enet.
      interop.net",Category="fabric_network_activity",Generator="Response:SlowScan",Type="NOTICE",Priority
      ="High",Body="Appliance=roach_motel.enet.interop.net,Reporting Segment=ENET network,Action=Response
      disabled,Response=Slow Scan,Duration=90seconds,Source Segment=Unprotected,Source IP=88.73.39.200,
      SourceMAC=00:01:30:BC:93:90,Current Target Count=0"
Apr 29 19:13:01 45.2.98.7 SentriantGenericAlert: Time="04/29/06 07:12 PM PDT",Host="roach_motel.enet.
      interop.net",Category="fabric_network_activity",Generator="Response:SlowScan",Type="NOTICE",Priority
      ="High",Body="Appliance=roach_motel.enet.interop.net,Reporting Segment=ENET network,Action=Response
      disabled,Response=Slow Scan,Duration=69seconds,Source Segment=Unprotected,Source IP=68.163.20.95,
      SourceMAC=00:01:30:BC:93:90,Current Target Count=0"
Apr 29 19:13:01 45.2.98.7 SentriantGenericAlert: Time="04/29/06 07:12 PM PDT",Host="roach_motel.enet.
      interop.net",Category="fabric_network_activity",Generator="Response:Slow...
```

Listing 8: Example Log Records

# Metrics

- A *counter* is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart
- A *gauge* is a metric that represents a single numerical value that can arbitrarily go up and down
- A *histogram* samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.
- Similar to a histogram, a *summary* samples observations (usually things like request durations and response sizes). While it also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

# Traces

- A *trace* is the complete processing of a request. The trace represents the whole journey of a request as it moves through all of the services of a distributed system.



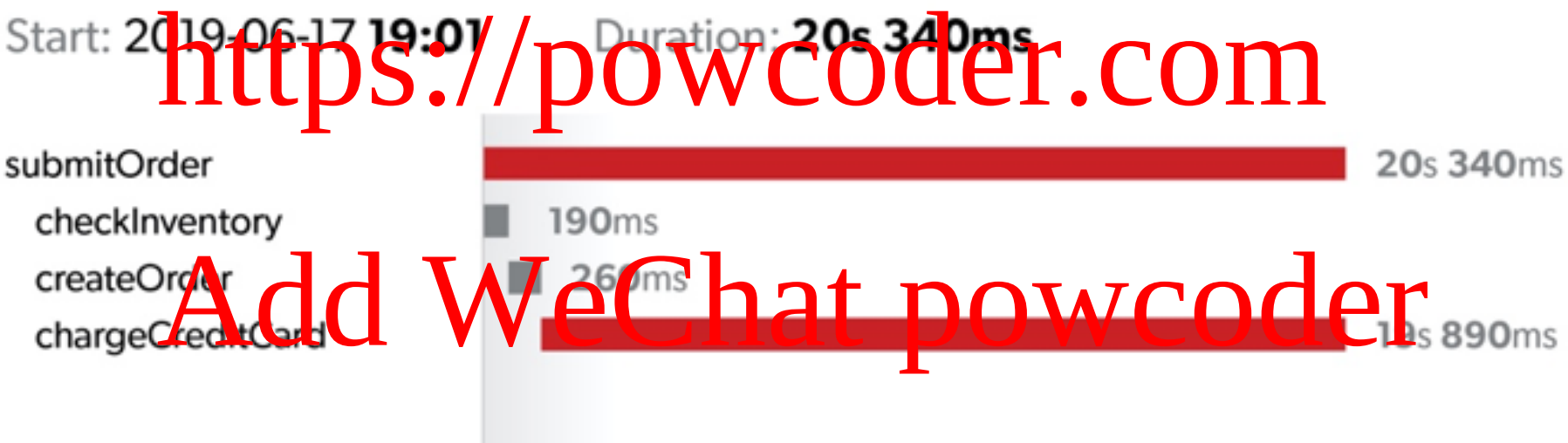



Figure: Trace Example

# Example Observability Pipeline

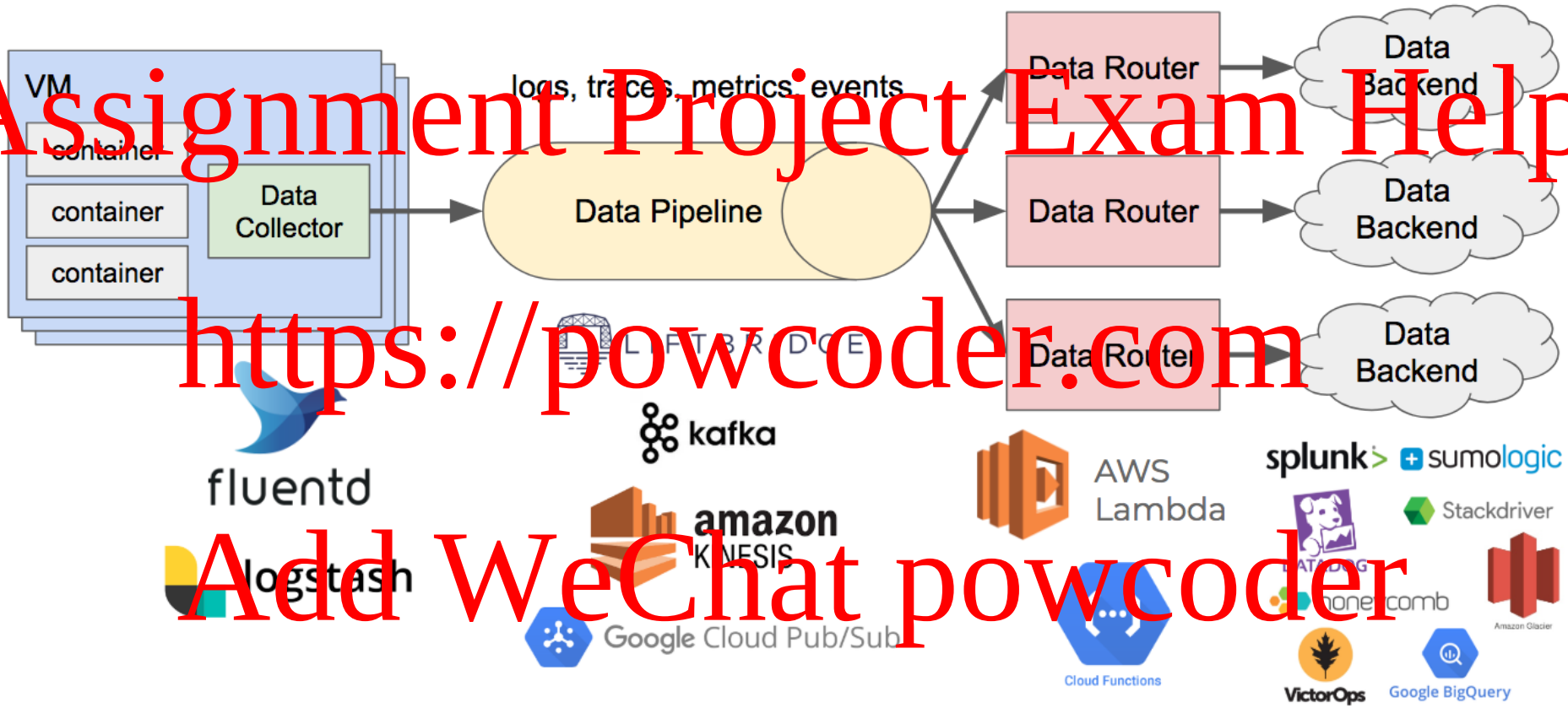





Figure: Example Observability Pipeline [Tre18]

# Dashboards

- *Dashboards* are the human-facing views into our systems that provide concise summaries of how the system is behaving by displaying time series metrics, logs, traces, and alarms data.
- Examples of types of dashboards:
  - *Customer experience dashboards* efficiently present metrics on overall service health and adherence to goals.
  - *System-level dashboards* contain enough data for operators to see how the system and its customer-facing endpoints are behaving. These dashboards primarily display interface-level monitoring data.
  - *Service instance dashboards* facilitate fast and comprehensive evaluation of the customer experience within a single service instance.
  - *Service audit dashboards* are used by operators to audit automated alarming across all service instances
  - *Capacity planning and forecasting dashboards* help us visualize the growth of our services

# Example Dashboard 1



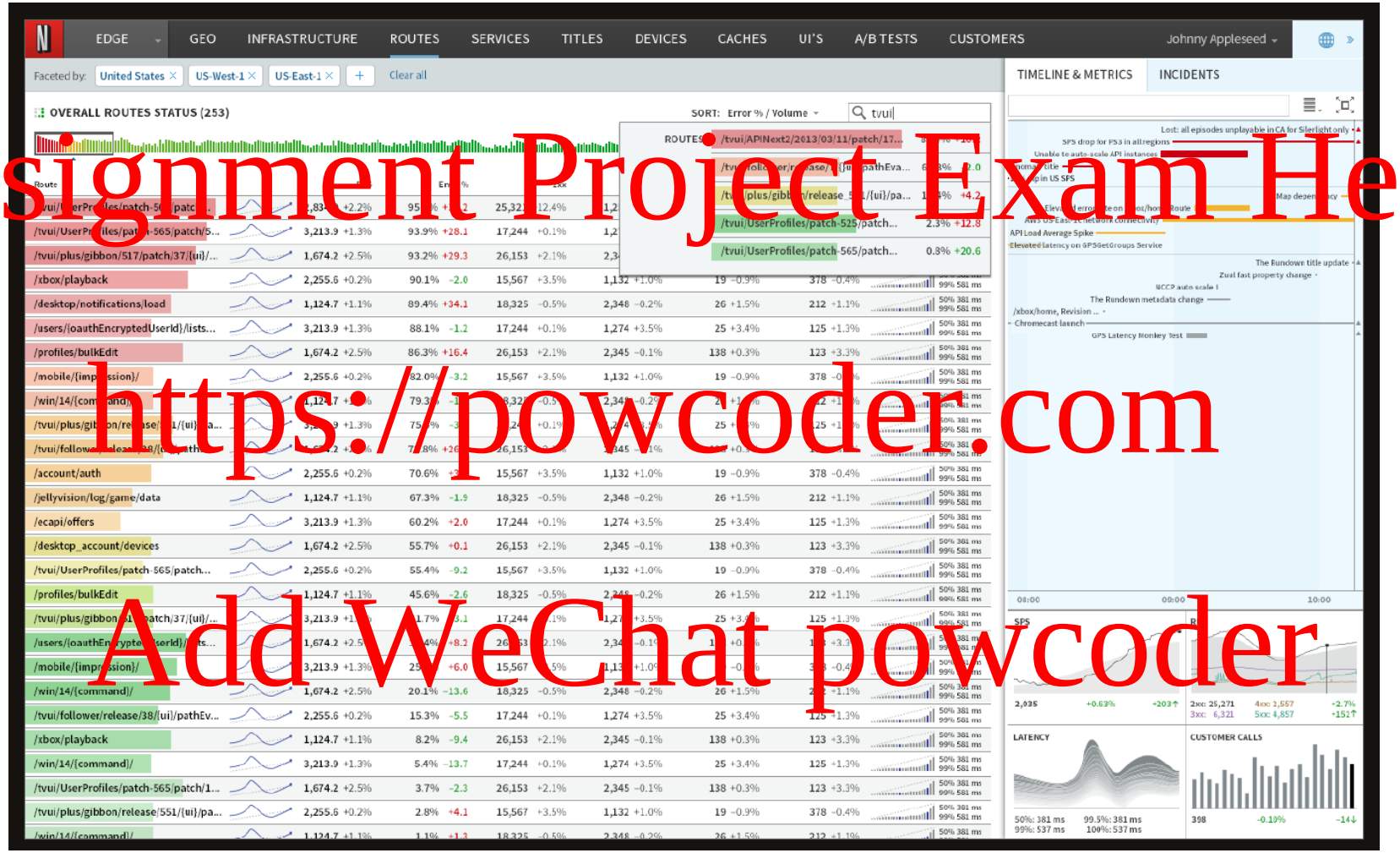



Figure: Netflix Route Dashboard [MBC14]

# Example Dashboard 2

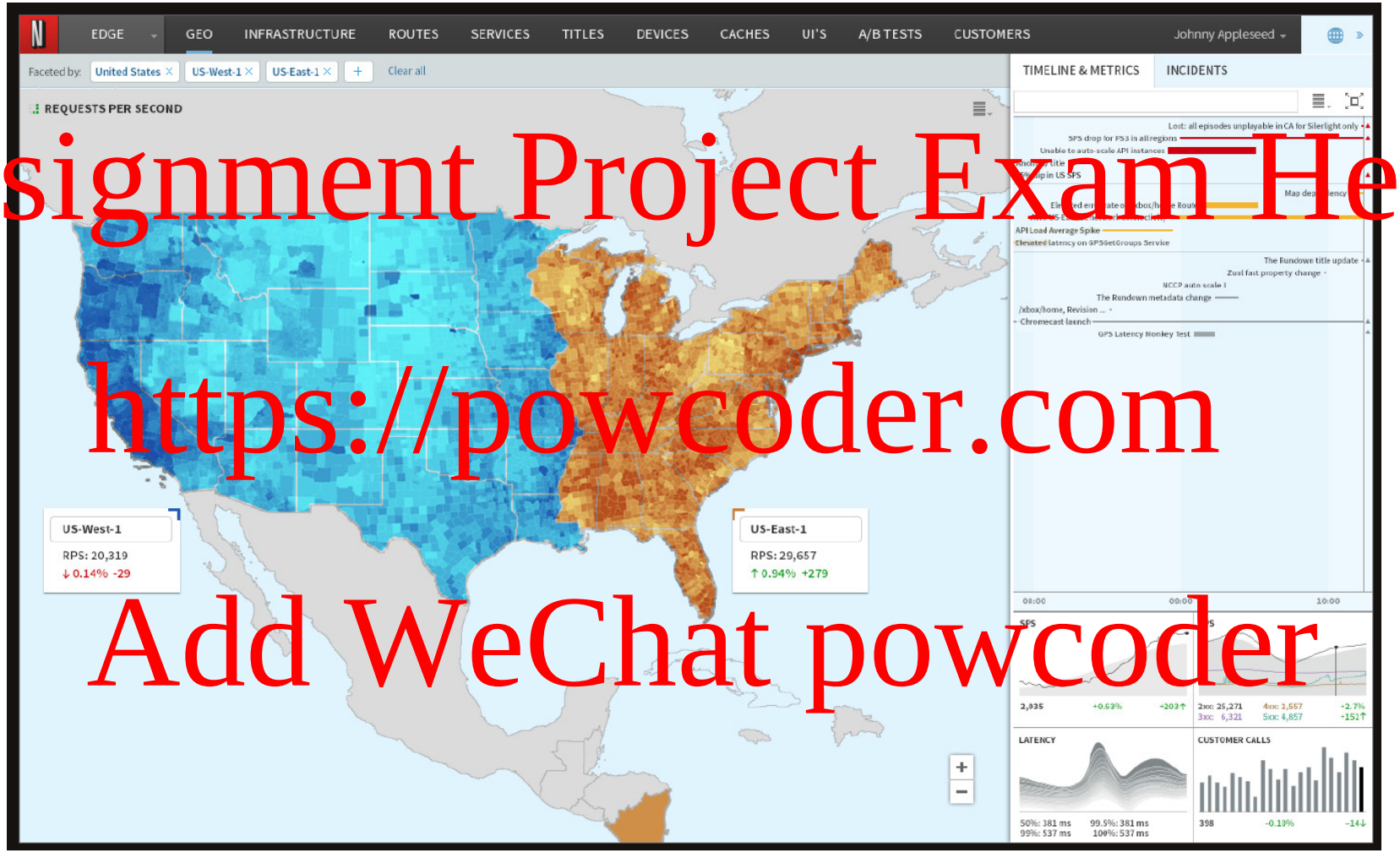





Figure: Netflix Geo Dashboard [MBC14]

# Example Dashboard 3



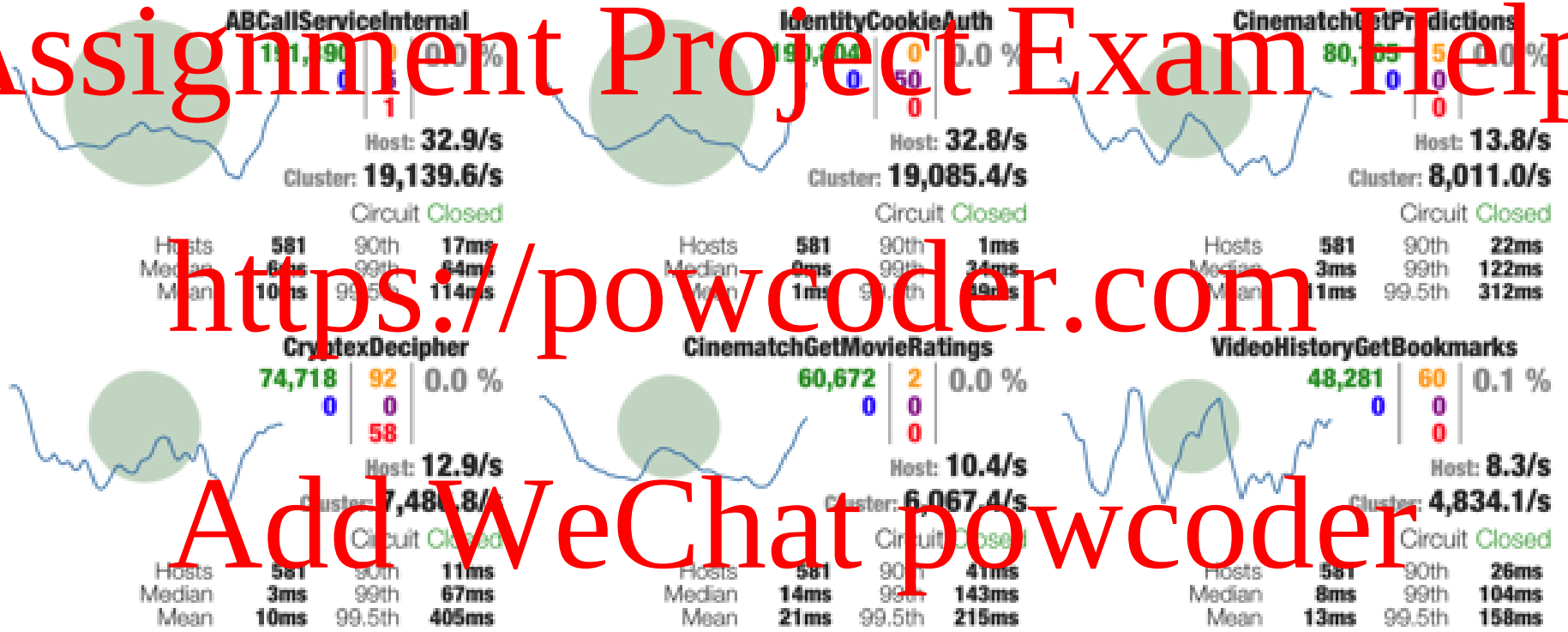



Figure: Netflix Service Dashboard [COS12]

# Example Dashboard 4

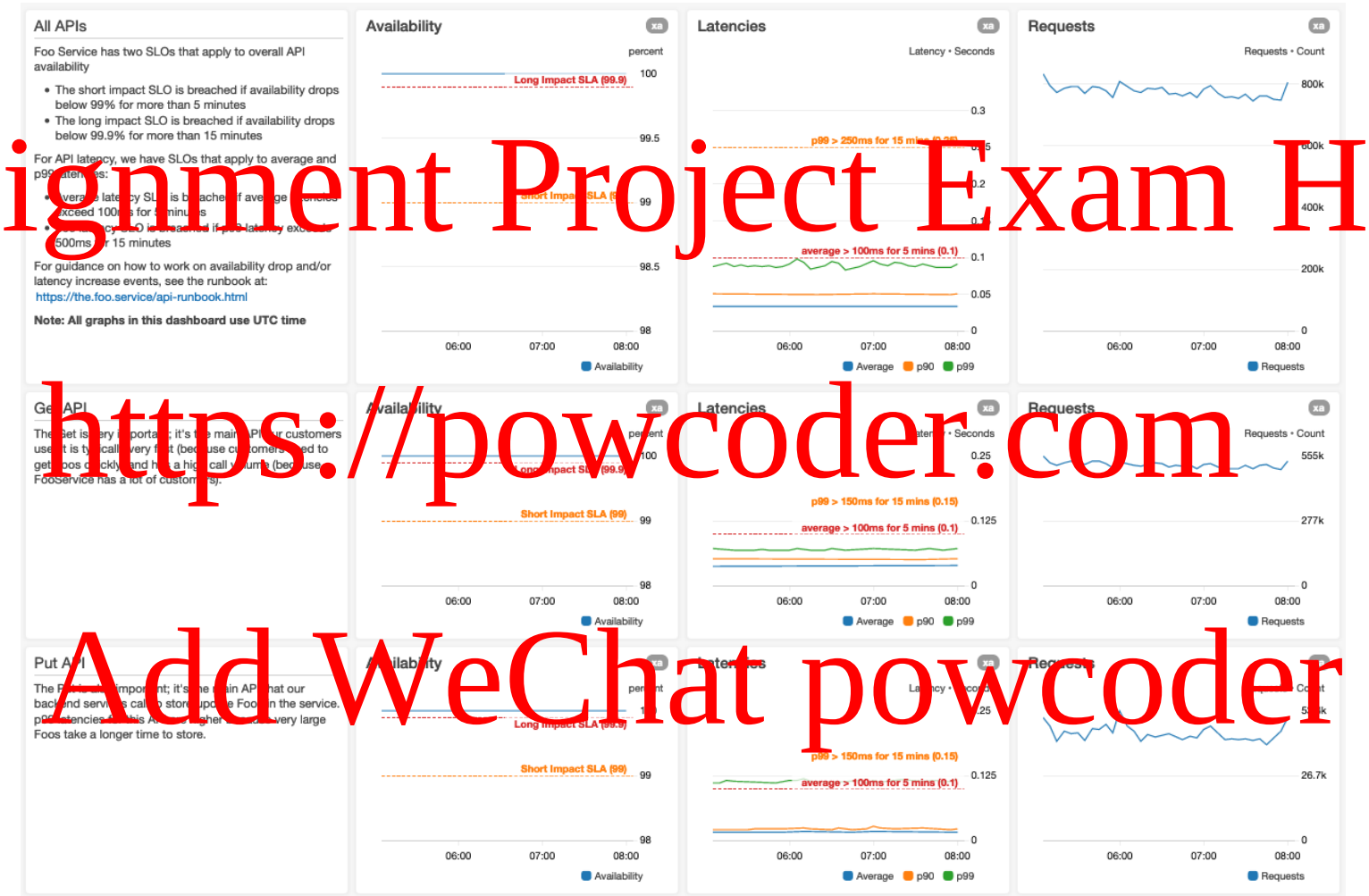



Figure: Hypothetical Amazon Service Dashboard

# Optional Readings

- The Observability Pipeline
- Understanding Observability: A Cloud Observability Framework
- Observability — A 3-Year Retrospective
- Tracing at Slack: Thinking in Causal Graphs
- Alerting on SLOs
- Building dashboards for operational visibility
- Telltale: Netflix Application Monitoring Simplified
- Debugging Incidents in Google's Distributed Systems
- GitOps Part 3 - Observability
- Monitoring and Tracing @Netflix Streaming Data Infrastructure
- High Resolution Performance Telemetry at Scale
- From Graphite To Prometheus — Things I've Learned
- How Deep Systems Broke Observability — and What We Can Do About It
- Microservice Observability, Part 1: Disambiguating Observability and Monitoring
- So You Want To Build An Observability Tool...
- Intro to Distributed Tracing

# Table of Contents

# Portability



- *Portability* is concerned with the ease with which the software that was built to run on one platform can be changed to run on a different platform
- For example, can your iOS mobile app run on your Android phone as well?
- Common techniques include:
  - Isolating dependencies to well-identified locations
  - Writing the software to run on a "virtual machine" (e.g. a hardware abstraction layer)

# Portability Example



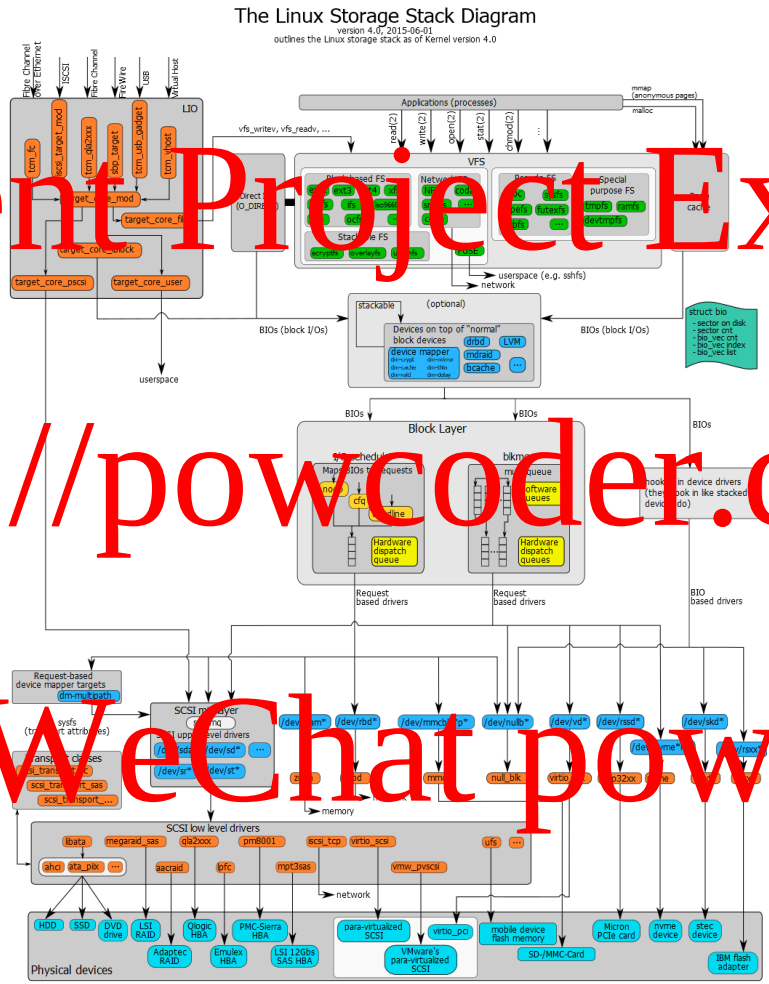



Figure: Linux Storage Stack [Wik18]

# Table of Contents

# Usability



- *Usability* is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides
- Common techniques include:
  - Cancel
  - Undo/redo
  - Pause/resume
  - Aggregate (allow user to group many objects together and perform an action on all of them)
- Don't forget to give the user continuous and useful feedback!

- Undo/redo is common enough that we should understand how to implement it
- There are a many ways to implement it, but a common one is to use the *Command* pattern and a stack of operations

# Undo/Redo Code Example

```
interface ICommand {
    void execute();
    void unexecute();
}

class InsertCharacterCommand : public ICommand {
    private TextBuffer _buf;
    private char _ch;

    public InsertCharacterCommand(TextBuffer buf, char ch) { _buf = buf; _ch = ch; }

    void execute() { _buf.push(_ch); }
    void unexecute() { _buf.remove(_buf.getLength()-1, 1); }
}

class Application {
    Stack<ICommand> _undoStack = ...;
    Stack<ICommand> _redoStack = ...;

    void undo() {
        ICommand cmd = _undoStack.pop();
        cmd.unexecute();
        _redoStack.push(cmd);
    }

    void redo() {
        ICommand cmd = _redoStack.pop();
        cmd.execute();
        _undoStack.push(cmd);
    }
}
```

# Table of Contents

- Homework 3 is due Thursday, October 22 at 5:30PM
- Quiz 3 is due Thursday, October 22 at 5:30PM

# References


[Bes18] Jonathan Besquin.
Mobile app architecture in a nutshell.
2018.

[COS12] Ben Christensen, Puneet Oberai, and Ben Schmaus.
Hystrix dashboard + turbine stream aggregator.
*The Netflix Tech Blog*, 2012.

[GS18] Josh Garverick and Willy Schaub.
Explore how to progressively expose your azure devops extension releases in production to validate, before impacting all users.
2018.

[LeS19] LeSS.
Feature teams.
2019.

[Lig20] Clare Liguori.
Automating safe, hands-off deployments.
2020.

[MBC14] Ranjit Mavinkurve, Justin Becker, and Ben Christensen.
Improving netflix's operational visibility with real-time insight tools.
*The Netflix Tech Blog*, 2014.

[Pok20] Sandeep Pokkunuri.
Ensuring rollback safety during deployments.
2020.

[Shr20] Ryan Shriver.
7 pipeline design patterns for continuous delivery.
2020.

[Tre18] Tyler Treat.
The observability pipeline.
2018.

[Wik18] Wikibooks.
The linux kernel/storage.
2018.

[Wik20a] Wikipedia.
Continuous delivery.
2020.

[Wik20b] Wikipedia.
Continuous deployment.
2020.

[Wik20c] Wikipedia.
Continuous integration.
2020.