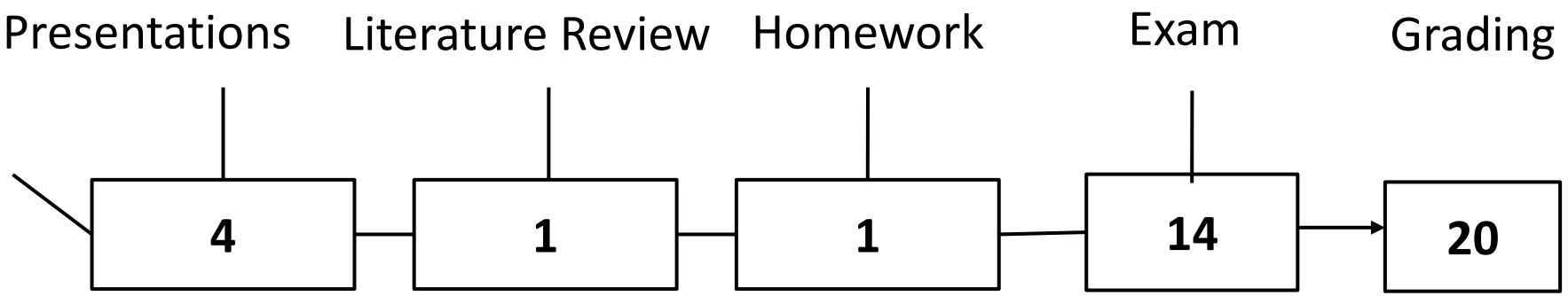
Large-Scale Systems

By Dr. Taghinezhad

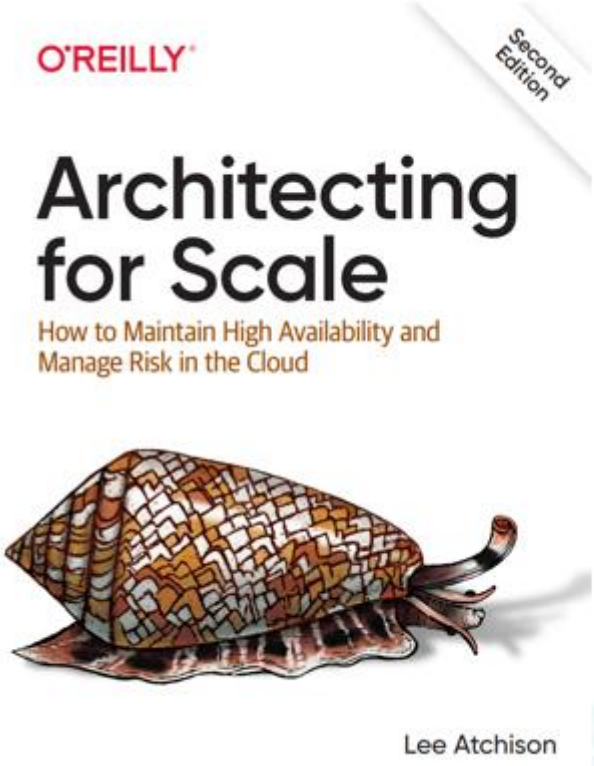
Mail:
a0taghinezhad@gmail.com

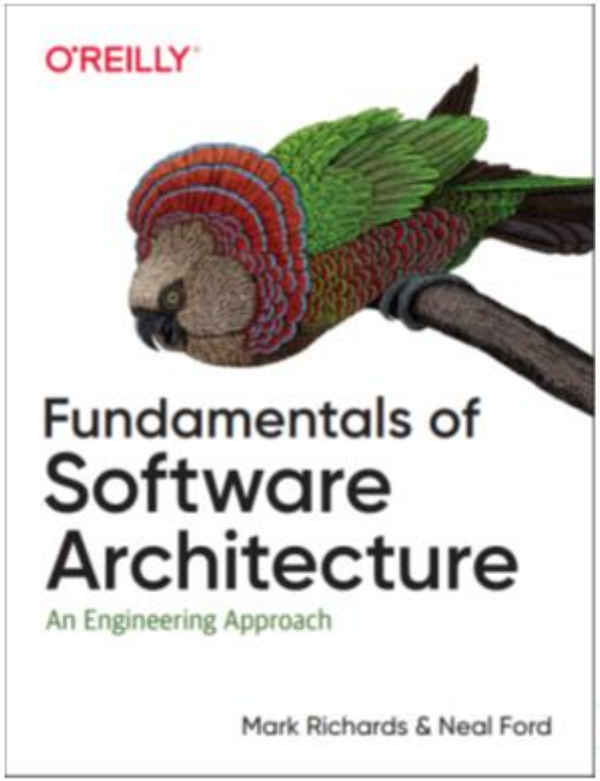
Scan for More Information:

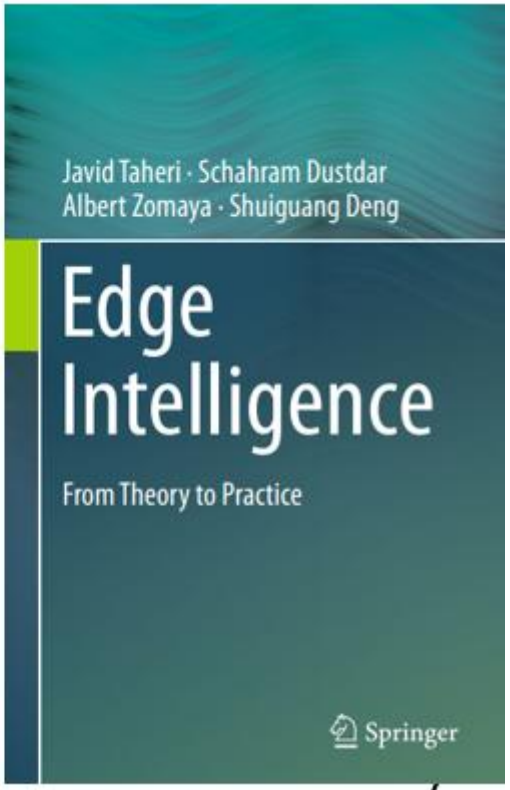
About This Course



References


O'REILLY
Second Edition
Architecting for Scale
How to Maintain High Availability and Manage Risk in the Cloud
Lee Atchison


O'REILLY
Fundamentals of Software Architecture
An Engineering Approach
Mark Richards & Neal Ford


Javid Taheri · Schahram Dustdar
Albert Zomaya · Shuiguang Deng
Edge Intelligence
From Theory to Practice
Springer

Projects include:

Each student must **review at least two papers** and write a **literature review** on these subjects. The subjects are as follows, but not limited to them:

1. Map-reduce- High Performance Distributed Computing (HPDC), System Reliability
2. Parallel processing, distributed computing systems, and computer networks
3. Cloud Computing, Fog Computing, Compute Continuum, Edge Intelligence

https://ataghinezhad.github.io/

Outlines of this course

- Availability: Maintaining Availability in Modern Applications

Chapter 1: Tenet 1. Availability: Maintaining Availability in Modern Applications

Availability: Maintaining Availability in Modern Applications

- In today's digital world, customers expect to be connected. They rely on services to be available, anytime, anywhere.

- Downtime is no longer acceptable. It leads to:
  - * **Frustration**: Customers abandon services that are unreliable.
  - * **Loss of Revenue**: Businesses lose money when their applications are unavailable.
  - * **Reputation Damage**: A single outage can tarnish a company's image.

Ex

Big game

- It's Sunday, the day of the big game. You've invited 20 friends over to watch it on your new 300-inch Ultra Max TV. But just as the game is about to start, the lights go out and the TV goes dark.
  - Disappointed, you call the local power company, and the representative says they guarantee only 95% availability of their power grid.
  - This is important because customers expect services to work all the time, and anything less than 100% availability can be catastrophic to your business.

The Importance of System Availability for Large-Scale Systems

- System availability is not just a **technical detail**;
  - it's a **fundamental business imperative** for large-scale systems.
- **Fundamental Questions** for All Companies:
  - * Why Buy From You? (Highlight the direct connection between availability and customer acquisition)
  - * What Do Your Customers Think? (Emphasize the impact of downtime on customer satisfaction and brand perception)
  - * How Do You Make Customers Happy? (Connect availability to business goals, revenue, and customer retention)

Defining Availability and Reliability

- **Reliability:**
  - The ability of a system to perform its intended operations **correctly** and **consistently** without errors.
  - **Example:** Software that always returns the correct answer to 2 + 3.
- **Availability:**
  - The ability of a system to be **operational** and **responsive** when needed.
  - **Example:** A system that **consistently returns an answer**, even if the answer is **sometimes** incorrect.
- A system can't be available if it's not reliable. (A system constantly returning wrong answers isn't useful).
- A system can't be reliable if it's not available. (A system that crashes frequently can't be relied upon).
- **Focus:** This course will emphasize building **highly available systems**, assuming your systems are reliable.

Reliability: Testing and Quality; Availability: Architecting for Resilience

- Reliability is often ensured through **thorough testing**.
  - **Test Suites:** We use test suites to catch software bugs and ensure that the system produces the correct output.
  - **Example:** A test suite for adding numbers would verify that 2 + 3 consistently results in 5.
- Availability is more challenging to achieve than reliability.
  - **Network Issues:** Even a reliable application can become unavailable due to network problems.
  - **This Course Focus:** We'll explore strategies to ensure your systems remain available despite network issues, hardware failures, and other challenges.

What Causes Poor Availability?

- As applications grow in popularity and usage, they may experience poor availability due to several factors:

  - 1. **Resource Exhaustion**: Increased **users** or data can **overwhelm** system **resources**, leading to slow or unresponsive applications.
  - 2. **Unplanned Load-Based Changes**: Rapid growth may necessitate quick code changes without adequate planning, increasing the risk of issues.
  - 3. **Increased Number of Moving Parts**: More developers and features can lead to complex interactions and potential conflicts within the application.
  - 4. **Outside Dependencies**: Reliance on external services (e.g., SaaS, cloud) makes applications vulnerable to availability issues from those resources.
  - 5. **Technical Debt:** As applications evolve, the accumulation of unresolved changes and bugs can increase complexity and the likelihood of problems.

These challenges can **manifest** gradually or **suddenly**, ultimately costing money, customer trust, and loyalty.

Measuring Availability

- Measuring availability is crucial for maintaining a highly available system.
  - By tracking availability, we can gain insights into our application's current performance and observe how it evolves over time.
- The most common method for assessing the availability of a system/application is to calculate the percentage of time it remains accessible to users.
  - This can be expressed using the following formula for a given period:
  - system availability

$$= \frac{\textbf{Total Seconds in Period} - \textbf{Seconds System is Down}}{\textbf{Total Seconds in Period}}$$

Example Calculation

- Let's consider a practical example. Suppose your website experienced two outages during April. The first outage lasted 37 minutes, and the second lasted 15 minutes. How much is the availability in Month?
- 1. **Calculate Total Downtime:**
  - - Total downtime = (37 minutes + 15 minutes) × 60 seconds/minute = 3,120 seconds
- 2. **Calculate Total Seconds in the Month:**
  - - Total seconds in April = 30 days × 86,400 seconds/day = 2,592,000 seconds
- 3. **Calculate Site Availability Percentage:**
  - - Using our formula:
  - Site Availability Percentage = 2,592,000s - 3,120s / 2,592,000s
  - - Site Availability Percentage = ≈ 99.8795%
- Even a small amount of downtime can significantly impact your overall availability percentage.
- Regularly measuring and analyzing availability is essential for ensuring that your application meets user expectations and remains reliable over time.

Understanding "The Nines" in Availability

- Availability is often expressed in terms of "the nines," which is a shorthand way to communicate high availability percentages.
- The number of nines indicates the level of reliability a system can achieve. Below is a summary of the availability levels, their corresponding percentages, and the allowable downtime in a **typical month**.

- Table 1-1: The Nines

- | Nines | Percentage | Monthly Downtime |
- |---|---|---|
- | 2 nines | 99% | 432 minutes |
- | 3 nines | 99.9% | 43 minutes |
- | 4 nines | 99.99% | 4 minutes |
- **| 5 nines | 99.999% | 26 seconds |**
- **| 6 nines | 99.9999% | 2.6 seconds |**

Example Analysis

- In your earlier example, your website achieved an availability of approximately 99.8795%, which falls short of **the 3 nines (99.9%) standard**.
  - This means that while your site is relatively reliable, it **does not meet** the expectations typically associated with **high-availability applications**.

- For a website aiming for 5 nines of availability, the requirement is extremely stringent: only 26 seconds of monthly downtime.
  - Achieving this level may necessitate advanced redundancy, failover mechanisms, and robust monitoring systems.

Determining Your Availability Goals

- Determining what constitutes a reasonable availability number depends on various factors:

  - 1. **System Type**: Critical applications (e.g., financial services) may require higher availability compared to less critical sites (e.g., personal blogs).

  - 2. **Customer Expectations**: Understand what your users expect in terms of uptime and reliability.

  - 3. **Business Needs**: Consider the impact of downtime on your revenue and brand reputation.

  - 4. **Industry Standards**: Research what competitors or similar businesses are achieving.

Determining Your Availability Goals

- Common Thresholds

  - For many **basic** web applications**, 3 nines** (99.9%) is often considered acceptable, allowing for about 43 minutes of downtime each month.
  - More **critical** applications may aim for **4 nines** (99.99%), while mission-critical systems might target 5 nines (99.999%) or even higher.

Understanding the Impact of Planned Outages on Availability

- Planned outages, while necessary for maintenance and updates, directly affect the overall availability of an application.
  - Many organizations mistakenly believe that regular **maintenance** windows do not **count against their availability metrics**. However, as illustrated in your example, they certainly do.

Ex

Example Calculation

- Here's a comment that I often overhear: "Our application never fails. That's because we regularly perform system maintenance. We keep our availability high by scheduling **weekly two-hour maintenance** windows and performing maintenance during these windows."
- Let's break down the example provided:
  - 1. Total Hours in a Week:
    - - 7 days × 24 hours = 168 hours
  - 2. Hours Unavailable Each Week:
    - - 2 hours (for planned maintenance)
  - 3. Calculating Availability:
    - - Availability Formula:
    - Availability = Total Hours in Period - Hours System is Down/Total Hours in Period
    - - Plugging in Values:
    - Availability = 168 hours - 2 hours/168 hours = 166/168≈ 0.9881 or 98.81%
- Without having a single failure of its application, the best this organization can achieve is 98.8% availability. This falls short of even 2 nines availability (98.8% versus 99%).

Implications for User Experience

- - **Negative User Experience**: Regardless of whether downtime is planned or unplanned if customers cannot access the application when they need it, their experience is negatively impacted.
- - Expectation Management: Customers expect high availability, and any downtime—planned or not—can lead to frustration, loss of trust, and potentially lost revenue.

Strategies to Mitigate Impact

- 1. **Schedule During Off-Peak Hours**: If possible, schedule maintenance during times when user activity is lowest to minimize impact.
- 2. **Communicate Clearly**: Inform users well in advance about scheduled maintenance windows and the expected downtime.
- 3. **Implement Redundancy**: Use load balancing and failover systems to allow for maintenance without affecting availability.
- 4. **Automate Maintenance Tasks**: Where feasible, automate maintenance tasks to reduce the duration of downtime.
- 5. **Monitor and Analyze**: Continuously monitor application performance and user feedback to adjust maintenance practices as necessary.

Improving Your Availability When It Slips

- Your application is operational and online.
  - Systems are in place, your team is efficient, and everything seems to be going well.
  - Traffic is increasing, and your sales team is pleased.
    - an unexpected outage occurs. Initially, it feels manageable—after all, your availability has been fantastic until now.
    - But then it happens again. And again.
- As outages accumulate, concerns grow. Your CEO starts to worry, customers ask questions, and your sales team becomes anxious. What was once a stable system is now facing instability, and the situation demands attention.

What to Do When Availability Begins to Slip

- If availability starts to decline, it's essential to take proactive steps to improve it and maintain customer satisfaction. Here are some key actions you can take:
- - **Measure and Track Current Availability**: Understanding your current availability is the first step. Regularly monitor and report your availability percentage to identify trends.
- - **Automate Manual Processes:** Streamlining processes can reduce human error and improve efficiency.
- - **Automate Deployment Processes:** This helps ensure consistency and speed in your updates.
- - **Maintain Configuration Management:** Keep track of all configurations in a management system for better oversight.
- - **Enable Quick Changes with Rollback Capabilities:** This allows for rapid experimentation without significant risk.
- **- Aim for Continuous Improvement:** Regularly assess and enhance your applications and systems.
- **- Prioritize Availability as a Core Issue:** Stay vigilant (observant) about availability as your application evolves.

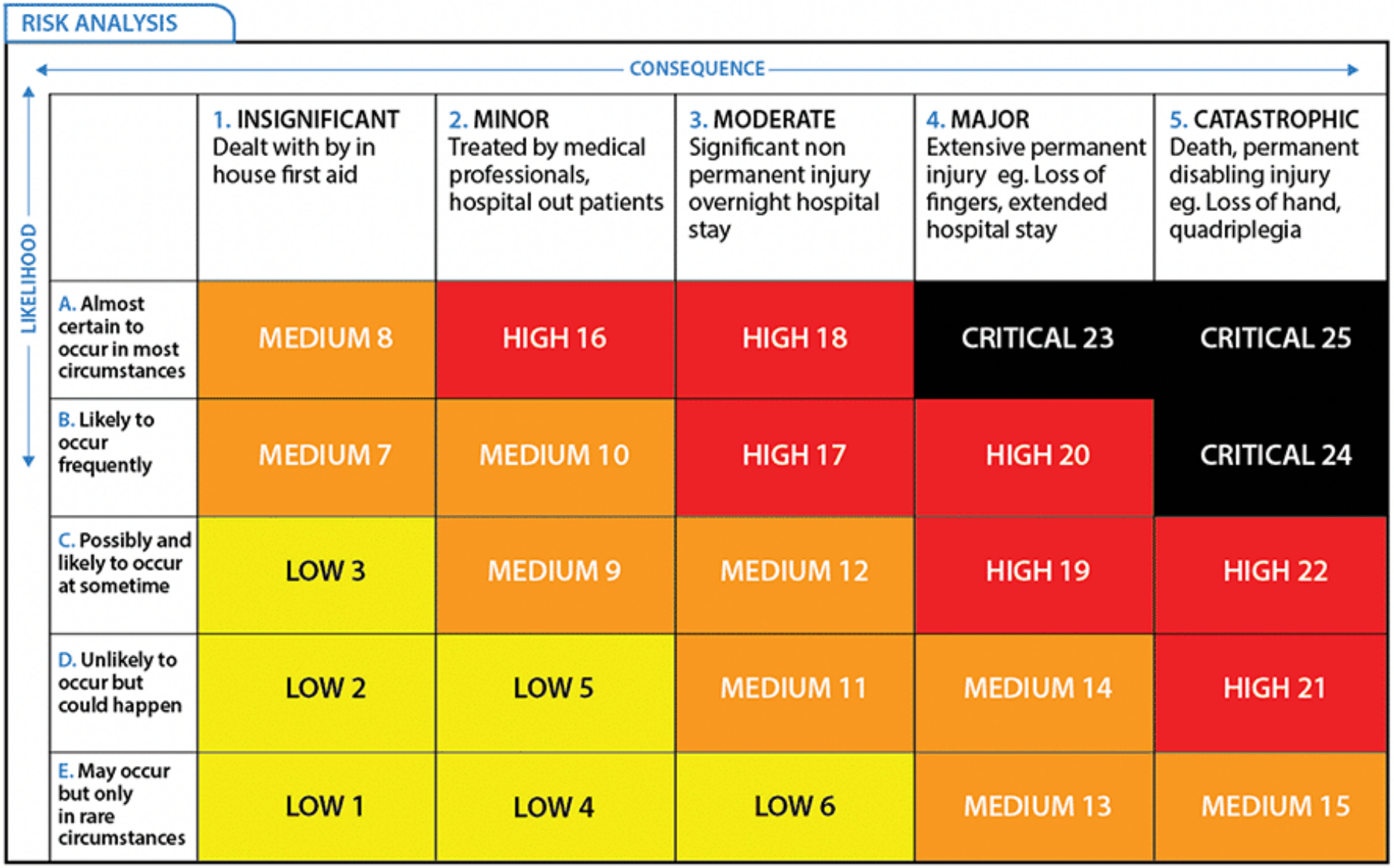
Detailed Steps for Improvement

- 1. Measure and Track Your Current Availability:
  - - Establish a baseline by tracking when your application is available or down.
  - - Calculate your availability percentage over time to measure performance.
  - - Monitor key events alongside availability data to identify correlations.

- 2. Utilize Service Tiers:
  - - Implement service tiers to categorize services based on their criticality to business operations.
  - - This distinction helps prioritize resources effectively.

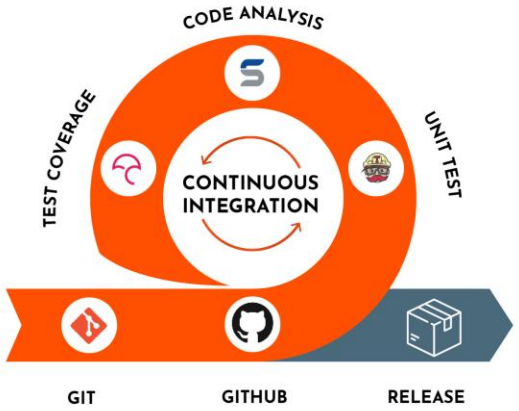
- 3. Create and Maintain a **Risk Matrix**:
  - - A risk matrix provides visibility into technical debt and associated risks within your application.
  - - Regularly review risk management plans and implement mitigation strategies.
- By following these steps, you can better manage your application's availability and avoid falling into a cycle of problems.

Risk Matrix

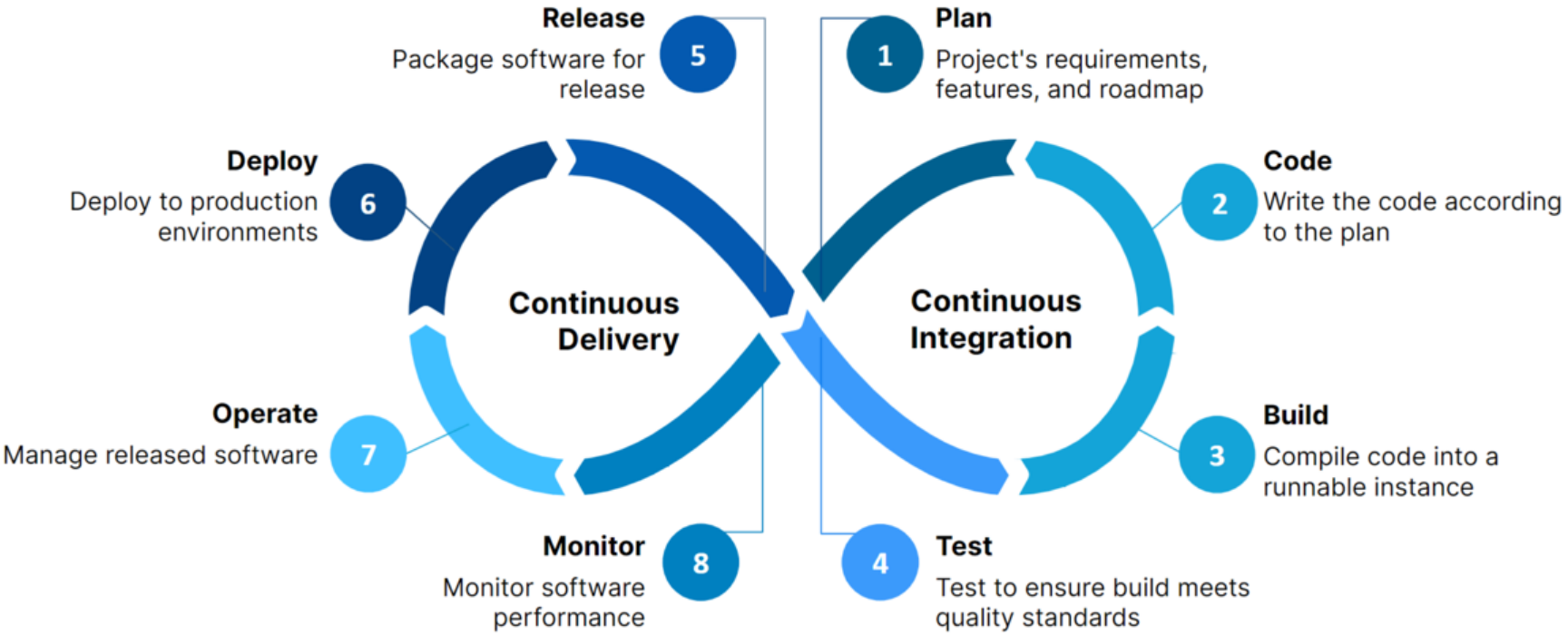

RISK ANALYSIS
CONSEQUENCE
LIKELIHOOD
1. INSIGNIFICANT Dealt with by in house first aid
2. MINOR Treated by medical professionals, hospital out patients
3. MODERATE Significant non permanent injury overnight hospital stay
4. MAJOR Extensive permanent injury eg. Loss of fingers, extended hospital stay
5. CATASTROPHIC Death, permanent disabling injury eg. Loss of hand, quadriplegia
A. Almost certain to occur in most circumstances
MEDIUM 8
HIGH 16
HIGH 18
CRITICAL 23
CRITICAL 25
B. Likely to occur frequently
MEDIUM 7
MEDIUM 10
HIGH 17
HIGH 20
CRITICAL 24
C. Possibly and likely to occur at sometime
LOW 3
MEDIUM 9
MEDIUM 12
HIGH 19
HIGH 22
D. Unlikely to occur but could happen
LOW 2
LOW 5
MEDIUM 11
MEDIUM 14
HIGH 21
E. May occur but only in rare circumstances
LOW 1
LOW 4
LOW 6
MEDIUM 13
MEDIUM 15

CI/CD

1. **Continuous Integration (CI)**:
    1. **What it is**: CI involves frequently integrating code changes into a shared repository. Developers commit their code to this central repository multiple times a day.
    2. **Why it matters**: By doing this, teams catch integration issues early. It ensures that everyone's work is combined and tested together, reducing the risk of conflicts and bugs during the later stages.
    3. **Example**: Imagine you're working on a team building a web application. Each developer writes their code and commits it to the central repository. The CI system automatically triggers a build process (compiling code, running tests, etc.) whenever a new commit is made. If any tests fail, the team is notified immediately, allowing them to fix issues promptly.



CI/CD



CI/CD

1. **Continuous Deployment (CD)**:
    1. **What it is**: CD extends CI by automating the deployment process. It ensures that code changes are automatically deployed to production environments once they pass all tests.
    2. **Why it matters**: CD reduces manual intervention, minimizes the time between writing code and deploying it, and increases the reliability of releases.
    3. **Example**: Continuing with our web application scenario, after successful CI, the CD pipeline takes over. It automatically deploys the tested code to staging or production servers. Users get access to new features or bug fixes without waiting for a manual deployment process.
2. **The CI/CD Pipeline**:
    1. The CI/CD pipeline is the set of automated steps that code goes through from development to production.

Ex

Scenario: ShopSmart E-commerce Platform

- ShopSmart has been enjoying high availability (99.9%) for several years. However, recent traffic spikes during holiday sales have led to unexpected outages, dropping availability to 98%. The CEO is concerned, and the customer support team is overwhelmed with complaints.
- Step-by-Step Implementation of Improvement Methods
- 1. Measure and Track Current Availability:
  - - Action: Implement monitoring tools like New **Relic** or **Datadog** to track uptime.
  - - Example: Set up alerts for downtime and log incidents in a shared dashboard. After a week, you find out that the platform has been down for a total of 12 hours due to server overload during peak times.
  - - Result: Calculate availability as follows:
  - Availability = (Total Time - Downtime/Total Time) × 100
  - For a week (168 hours):
  - Availability = (168 - 12/168) × 100 = 92.86%

Ex

Scenario: ShopSmart E-commerce Platform

- **Utilize Service Tiers**:
  - - Action: Categorize services into tiers: critical (checkout process), important (product catalog), and non-critical (user reviews).
- - Example: Allocate more resources (like dedicated servers) to the checkout process during high traffic times.
  - - Result: During the next sale, the checkout process remains functional even when the product catalog experiences slowdowns.

What is CI/CD?

CI/CD = Continuous Integration + Continuous Deployment

🎯 **Goal:** Deliver high-quality code faster and safely.

Term	Meaning	Example
CI	Automatically test and check code whenever it changes	Run tests when you push to GitHub
CD	Automatically deploy the tested code to production	Update your website automatically

CI/CD Workflow Overview

- 1 Developer writes code
- 2 Pushes to GitHub
- 3 GitHub Actions runs tests
- 4 If all tests pass → Build Docker image
- 5 Deploy app to cloud server

- **Project Structure Example**
  - my_python_app/
  - ├── .github/workflows/ci-cd.yml
  - ├── app/
  - │ └── main.py
  - ├── requirements.txt
  - ├── Dockerfile
  - └── tests/
  - └── test_main.py

Tools Used in Python CI/CD

Tool	Purpose
Git & GitHub	Store and manage your code
GitHub Actions	Runs automated CI/CD pipelines
pytest	Runs tests automatically
flake8 / black	Check and format your code
Docker	Package your app for deployment
AWS / Heroku / Render	Host your app online

CI Example – Testing and Linting

.github/workflows/ci-cd.yml

```
name: Python CI

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.10"
      - run: pip install -r
requirements.txt pytest flake8
      - run: flake8 app
      - run: pytest
```

CD Example – Deploying with Docker

After tests pass:

```
deploy:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: docker/login-action@v3
      with:
        username: ${{ secrets.DOCKER_USER }}
        password: ${{ secrets.DOCKER_TOKEN }}
    - uses: docker/build-push-action@v6
      with:
        push: true
        tags: yourname/my_python_app:latest
```

Dockerfile Example

Dockerefile:
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app/main.py"]

Packages your code so it runs the same anywhere — local or cloud.

Dockerfile Example

Step	Stage	Tool
1	Write code	VSCode / PyCharm
2	Push to GitHub	Git
3	Test automatically	GitHub Actions + pytest
4	Build container	Docker
5	Deploy online	AWS / Render / Heroku

Scenario: ShopSmart E-commerce Platform

- 3. **Create and Maintain a Risk Matrix:**
  - - Action: Develop a risk matrix to identify potential failure points, such as server overload, database latency, or third-party service outages.
  - - Example: Identify that the database is a single point of failure. The risk matrix shows it has a high likelihood of causing downtime during peak traffic.
  - - Result: You implement a read-replica database solution to distribute load and reduce risk.

- 4. Automate Manual Processes:
  - - Action: Use CI/CD tools like Jenkins or **GitHub Actions** to automate deployment processes.
  - - Example: Before the holiday season, set up automated deployment for minor updates to ensure quick rollouts without manual intervention.
  - - Result: Updates are deployed smoothly with no downtime, improving overall system reliability.

Ex

Scenario: ShopSmart E-commerce Platform

- 5. Enable Quick Changes with Rollback Capabilities:
  - - Action: Implement **feature flags** to allow quick rollbacks in case of issues.
  - - Example: During a major update, if users report issues with a new payment method, you can quickly disable it without affecting other functionalities.
  - - Result: This minimizes customer impact and maintains user trust.
- 6. Aim for Continuous Improvement:
  - - Action: Schedule regular review meetings to assess performance metrics and discuss potential improvements.
  - - Example: After each major sale, analyze what worked and what didn't. Adjust your infrastructure accordingly.
  - - Result: Continuous feedback leads to proactive measures, such as increasing server capacity before anticipated traffic spikes.

Scenario: ShopSmart E-commerce Platform

- 7. Prioritize Availability as a Core Issue:
  - - Action: Make availability a key performance indicator (KPI) for all teams.
  - - Example: Include uptime targets in team objectives and tie some performance reviews to these metrics.
  - - Result: All teams become more aware of their role in maintaining availability, fostering a culture of accountability.

By systematically implementing these methods, ShopSmart not only improved its availability from 98% back to 99.9% but also built a more resilient system capable of handling future growth. Regular monitoring and proactive measures ensured that outages became less frequent, leading to increased customer satisfaction and trust.

Automate Your Manual Processes

- Avoid Manual Operations in Production:
  - - Performing manual changes in a production environment can lead to unpredictable results—either improvements or compromises.
  - Benefits of Using Repeatable Tasks:
    - 1. Testing Before Implementation:
      - - Allows you to test tasks before applying them, helping to avoid mistakes that could lead to outages.
    - 2. **Task Customization**:
      - - You can tweak tasks to meet specific requirements, enabling improvements before implementation.

    - 3. **Peer Review:**
      - - Having tasks reviewed by a third party increases the likelihood of identifying potential side effects.

Automate Your Manual Processes

- 4. **Version Control:**
  - - Track changes to tasks: who made them, when, and why. This enhances accountability and traceability.
- 5. **Consistency Across Resources:**
  - - Apply the same change consistently across all affected servers, improving overall system coherence.
- 6. **Operational Consistency:**
  - - Avoid "one-off" changes that lead to server drift, making diagnostics more challenging. Consistent changes create a reliable operational baseline.
- 7. **Auditable Tasks:**
  - - Repeatable tasks can be analyzed later for their impact, whether positive or negative.

Automate Your Manual Processes

- **Security Considerations:**
  - - Many systems restrict access to production environments, allowing only automated processes and procedures to interact with them. This is a deliberate strategy to maintain stability and security.
- **If a task cannot be repeated reliably**, it loses its utility. Implementing repeatable tasks is crucial for maintaining stability in your systems and applications.

Ex

Example: Database Schema Changes:Avoid Manual Operations in Production

- - Scenario: A **developer needs to add a new column** to a production database table.
- - Risk: If they manually execute the SQL command without proper testing, it could lead to downtime, data corruption, or unexpected application behavior
  - Benefits of Using Repeatable Tasks:

- 1. **Testing Before Implementation:**
  - - Approach: Create a migration script that adds the new column. This script is first executed in a staging environment.
  - - Benefit: Testing in staging ensures that the script works as intended without affecting production. Any issues can be identified and resolved beforehand.
- 2. **Task Customization:**
  - - Approach: The migration script can be adjusted to include default values or constraints based on specific application requirements.
  - - Benefit: Customizing the task allows for improvements that align with business needs before it's applied to production.
- 3**. Peer Review:**
  - - Approach: The migration script is submitted for review by another developer or database administrator.
  - - Benefit: Peer review increases the likelihood of catching potential issues, such as performance impacts or unintended consequences of the schema change.

Example: Database Schema Changes

- 4. **Version Control:**
  - - Approach: Store the migration scripts in a version control system (e.g., Git).
  - - Benefit: This allows tracking of who made changes, when they were made, and the rationale behind them, enhancing accountability.
- 5. **Consistency Across Resources:**
  - - Approach: Use a migration tool (e.g., Flyway or Liquibase) to apply the same schema change across multiple database instances (e.g., dev, staging, production).
  - - Benefit: Ensures that all environments are consistent, reducing the likelihood of environment-specific issues.
- 6. **Operational Consistency:**
  - - Approach: By using repeatable migration scripts rather than manual changes, you avoid ad-hoc modifications.
  - - Benefit: This creates a reliable operational baseline and prevents server drift, making it easier to diagnose issues in the future.
- 7. **Auditable Tasks:**
  - - Approach: Maintain a log of all executed migration scripts and their outcomes.
  - - Benefit: This allows for later analysis of the impact of changes, whether positive (e.g., improved performance) or negative (e.g., increased load times).

Automated Deployments

- Consistency: Automating deployments ensures that changes are applied uniformly across your system. This means you can confidently apply similar changes later with predictable results.

- - Reliable Rollbacks: Automated deployment systems enhance the reliability of rolling back to known good states, minimizing downtime and risk

Maintain Configuration Management:

- **Automated Changes:** Instead of manually tweaking configuration variables, implement a process to automate these changes. At a minimum, create a script for the change and check it into your software change management system.

- **-Uniformity Across Servers:** This approach allows for consistent application of changes across all servers in your system. When adding or replacing servers, having a known configuration improves safety and minimizes impact

Infrastructure as Code (IaC)

- Modern Best Practices: Embrace **Infrastructure as Code** by **describing your infrastructure in a machine-readable format.**
  - Use tools like Puppet or Chef to automate the **creation** and **updating** of your infrastructure based on this specification.
- - **Version Control**: Store your infrastructure specifications in a version control system. This allows you to track changes just like you would with code, ensuring accountability and traceability.
- **- Change Management**: Any **infrastructure** or **configuration** change must go through the specification:
  - 1. Ensure **consistent**, stable configurations across all components.
  - 2. Track all changes for rollback capabilities and correlation with system events.
  - 3. Implement a **peer review process** for infrastructure changes, akin to code reviews.
  - 4. **Create duplicate environments** for testing, staging, and development, mirroring production.

Comprehensive Application

- - All Components: This process applies not only to **servers** but also to **cloud components**, (virtual network) VPCs**, load balancers, routers, and monitoring systems.**
- - No Exceptions: For IaC to be effective, it must be consistently applied to all system changes. Bypassing the management system is never acceptable.

Real-World Example

- - Operational Pitfalls (unforeseen): Consider a scenario where an operational update states, “We had a problem with one of our servers last night. I tweaked the kernel variable and increased the maximum number of open files.”
- - This fix may work temporarily but can lead to inconsistencies across servers and undocumented changes that cause future issues.

Five Focus Areas to Improve Application Availability

- Building a scalable application with high availability is challenging. Issues can arise unexpectedly, causing disruptions for users. Here are five key focuses to ensure your application remains available:

Five Focus Areas to Improve Application Availability

- 1. Build with Failure in Mind
  - - Anticipate Failures: As Werner Vogels, CTO of Amazon, states, “Everything fails all the time.”
  - **1) Design Considerations:**
    - - Error Handling: Implement error-catching mechanisms.
    - - Example:
      - Use **retry logic** to attempt to reconnect to a failed service before giving up.
  - **2) Dependencies:**
    - Circuit breaker patterns are particularly useful for handling dependency failures because they can reduce the impact a dependency failure has on your system. Without a circuit breaker, you can decrease the performance of your application because of a dependency failure
  - **3) Customers:** What do you do when a component that is a customer of your system behaves poorly

Five Focus Areas to Improve Application Availability

- **2) Always Think About Scaling**
  - Prepare for Growth: Design your system to handle increased loads without performance degradation.
    - New server must be added to your system easily.
    - - Example: Use load balancers to distribute traffic evenly across servers.
  - Reload only dynamic parts and use (CDN) and browser from static parts. Don’t load the entire website from sever every time.

Five Focus Areas to Improve Application Availability

- **3) Mitigate Risk**
  - - Identify Vulnerabilities: Assess components that could fail and develop strategies to minimize their impact.
  - - Example: Utilize redundancy for critical services (e.g., multiple database replicas).
    - Imagine you are selling a product on a website, and your search system is a different system, if it fails, all your system fails. How would you respond to that?
      - One way is to show your most popular product and respond to apologize for the search problem.

Five Focus Areas to Improve Application Availability

- 4) **Monitor Availability:** You can't know if there is a problem in your application unless you can see a problem
  - **Server monitoring** To monitor the health of your servers and make sure they keep operating efficiently.
  - **Configuration change monitoring:** To monitor your system configuration and identify if and when changes to your infrastructure impact your application.
  - **Application performance monitoring:** To look inside your application and services to make sure they are operating as expected.
  - **Synthetic testing:** To examine in real time how your application is functioning from the perspective of your users, in order to catch problems customers might see before they actually see them.
  - **Alerting:** To inform appropriate personnel when a problem occurs so that it can be quickly and efficiently resolved, minimizing the impact on your customers.

Five Focus Areas to Improve Application Availability

- **5) Respond to Availability Issues Predictably**
  - Unresponsive Services:
    - When a service becomes unresponsive, several remedies can be employed:
      - Run diagnostic tests to identify the issue.
      - Restart a known problematic daemon.
      - Reboot the server if necessary.
  - Standardized Failure Handling:
    - Implement standard procedures to reduce system downtime.
    - These processes aid in identifying the root cause of recurring issues.
    - Service owners must be alerted first to address problems
    - Related teams should also receive alerts to prepare for potential impact on their dependent services.

Five Focus Areas to Improve Application Availability

- Real-World Example: Icon Failure
  - A past project experienced a major outage due to a third-party icon generation system failure. The application assumed this dependency would always function. When it failed, the entire application went down.
- Lessons Learned:
  - - Anticipate Dependency Failures: Implement logic to handle such failures gracefully.
  - - Error Recovery: Detect the failure and either remove the icon or catch the error, allowing the rest of the application to function normally.

End of Chapter 1