# Thinking DevOps

—--------------------------------------------

This module is all about thinking DevOps and getting results! You will learn how social coding as a community creates better code. You will examine how creating shared repositories and pair programming results in defects found earlier and a broader understanding of the code base for the team. You will see how working in small batches reduces waste and results in delivering useful applications quickly. You will discover how producing a minimum viable product allows you to test a hypothesis and gain valuable feedback about delivering what the customer really desires. Test driven development will allow you to develop faster and with more confidence. Behavior driven development results in improved communication and more meaningful information from your stakeholders. You will learn how cloud native architecture is used to build resilience in your product and take advantage of horizontal scaling and independently deployable microservices. Failures will happen. You will be able to take advantage of tools that are designed to help you recover quickly from failures.

### **Learning Objectives**

* Describe social coding principles
* Describe the benefits of working in small batches
* Define minimum viable product
* Describe test driven development
* Describe behavior driven development
* Describe key advantages of using cloud native microservices
* Describe how designing for failure benefits the development process

—----------------------------------------------------------------------------------------------------------------------------

### Video 1: **Social Coding Principles**

**—-------------------------------------------------------**

### **Summary of the Lecture on Social Coding and Pair Programming**

**Key Concepts:**

1. **Social Coding:**
   * Defined as "Open Source for Inner Source."
   * A shift from private repositories to public repositories within enterprises, encouraging collaboration on internal projects.
   * Breaks down barriers of access control, allowing for code reuse and collaboration among teams.
   * Prevents the common issue of teams reinventing solutions due to a lack of awareness of existing code.
2. **Benefits of Social Coding:**
   * Encourages contributions from all team members, increasing visibility of ongoing work.
   * Promotes discussion between developers and repository owners, leading to collaborative feature development.
   * Enhances resource efficiency by reusing existing code instead of rewriting it, saving time and costs.
3. **Pair Programming:**
   * A practice where two programmers share a single workstation (one driver, one navigator).
   * Roles are swapped periodically (every 20 minutes) to foster collaboration.
   * Supports social coding principles by improving code quality and knowledge transfer.

**Advantages of Pair Programming:**

* **Higher Code Quality:**
  + "Programming out loud" leads to better understanding and earlier detection of defects.
  + Explaining code helps identify bugs that may be missed during solo coding.
* **Skills Transfer:**
  + Pairing junior developers with seniors facilitates learning and sharing of best practices.
* **Better Code Understanding:**
  + Two sets of eyes on the code prevent issues related to code only being understood by one person, ensuring maintainability.

**Conclusion:**

* Social coding and pair programming enhance collaboration, improve code quality, and foster a culture of shared learning within development teams. By leveraging existing resources and encouraging contributions, companies can create a more efficient and innovative work environment.

—---------------------------------------------------------------------------------------------------------

### Video 2: **Git Repository Guidelines**

—--------------------------------------------------------------------------------

### **Summary of the Lecture on Git Feature Branch Workflow and Social Coding**

**Key Concepts:**

1. **Repository Guidelines:**
   * **One Component per Repository:** Each microservice or component should have its own repository. Avoid using a "mono repo" (a single repository with multiple services), as this forces people to deal with unnecessary code. It's best to have separate repos for better focus and clarity.
2. **Git Feature Branch Workflow:**
   * **Use Feature Branches:** For each issue, bug, or feature you work on, create a new branch. Branches are lightweight in Git, and it's better to create short-lived branches rather than maintaining a long-running "development" branch.
   * **Delete Feature Branches:** Once the work is done, the feature branch should be deleted, keeping the workflow clean and organized.
   * **Master Branch:** The master branch remains the central branch, and all changes must go through feature branches and pull requests.
3. **Pull Requests:**
   * **Code Review Process:** The only way code should enter the master branch is through a pull request. This ensures that other team members can review the code before it is merged.
   * **No Self-Merging:** The author of the pull request should not merge their own code. Another team member must review and merge it, ensuring that two sets of eyes review every piece of code before it enters the master branch.

**Workflow Overview:**

1. Create a repository for your component or fork an existing one.
2. Clone the repository to your local workstation.
3. Create a feature branch for your work (new features, bug fixes, etc.).
4. remote branch when ready.
5. Submit a pull request for review and, after approval, have the code merged into the master branch.

**Conclusion:**

* Following the **Git Feature Branch Workflow**—with separate repositories, feature branches, pull requests, and code reviews—ensures code quality, collaboration, and better organization. This process aligns with social coding principles by promoting transparency, collaboration, and code reuse.

—----------------------------------------------------------------------------------------------------------------------------

### Video 3: **Working in Small Batches**

—------------------------------------------------------------------------------------

**Key Concepts:**

* **Working in Small Batches**: A Lean Manufacturing principle that helps teams learn quickly from their decisions, avoid waste, and improve based on faster feedback.
* **Single Piece Flow**: Performing each step of a process one by one to receive immediate feedback rather than waiting for all steps in large batches to complete.
* **DevOps Application**: Aligns with DevOps practices like Continuous Integration and Continuous Delivery by allowing faster releases and feedback.

**Benefits of Small Batches:**

* **Faster Feedback**: Small batches allow for rapid feedback, enabling quicker adjustments and avoiding costly errors.
* **Waste Reduction**: Minimizes wasted time by preventing development of features that customers don’t want.
* **Better Quality Control**: Early detection of defects and errors (e.g., typos, missing glue) helps address issues before they affect a large batch.

**Real-Life Example:** Consider a marketing team mailing brochures. In a **large batch process**, folding 50 brochures takes 5 minutes, then inserting, sealing, and stamping each step adds more time, totaling 16 minutes before they inspect the first product. If there’s an error (e.g., typo), they’ve wasted 16 minutes and would need to restart. In contrast, a **small batch approach** (single piece flow) takes only 24 seconds to complete one brochure, allowing inspection and immediate detection of any mistakes.

**Application in Software Development**: In DevOps, small batches could mean dividing software features into smaller tasks that can be delivered in increments rather than waiting for all features to be complete. This helps ensure feedback from users or testers happens early, and developers can pivot if needed. For example, instead of waiting weeks for feedback, releasing part of a feature in a day allows developers to make adjustments without wasting time building a product customers don’t need.

### **How it's useful:**

Working in small batches improves efficiency in many industries, particularly in areas requiring fast feedback and flexibility. This practice is crucial in DevOps environments where rapid, continuous improvements are essential for timely software delivery.

### **Real-Life Examples:**

1. **Tech Startups**: Releasing software features in small batches (e.g., beta versions) to gather user feedback, correct errors, and pivot as needed to meet market demands.
2. **Manufacturing**: A car manufacturer may test a single new part before rolling it out in large production, allowing early detection of defects.
3. **Marketing**: A team mailing event invites can ensure the quality of their mailing materials by testing one invite before producing and sending thousands.

—----------------------------------------------------------------------------------------------------------------------------

### Video 4: **Minimum Viable Product**

—--------------------------------------------------------------------------------

**Key Concepts:**

* **Minimum Viable Product (MVP)**: A minimum viable product is not phase one of a project or a beta version. It is the smallest version of a product you can create to test a hypothesis and gather feedback from customers to learn whether to continue (persevere) or change direction (pivot).
* **Focus on Learning**: The purpose of an MVP is not just to deliver a product but to learn from the customer’s feedback and improve the product iteratively.

**MVP Process:**

* **Iterative Development**: An MVP allows for multiple iterations, where each version tests a new hypothesis or feature and gathers feedback from the customer.
* **Learning Over Delivery**: The goal of an MVP is to learn about customer needs and preferences, which may evolve as they interact with early versions of the product. This helps ensure the final product is what the customer truly wants.
* **Pivot or Persevere**: After each MVP iteration, teams decide whether to pivot (change direction) based on feedback or persevere (continue building on the current approach).

**Real-Life Example:**

1. **Incorrect MVP Execution**: Imagine a team building a car for a customer. In their first iteration, they deliver a wheel. In the next iteration, a chassis, and later a car without a steering wheel. The customer gives no feedback until the car is complete, and the team only discovers at the end whether the customer likes the car. This is not an effective MVP because there is no learning at each step.
2. **Effective MVP Execution**: In another example, the team starts by giving the customer a skateboard, asking if the red color is right. The customer provides feedback, and in the next iteration, they receive a scooter with steering, then pedals for speed, eventually evolving into a convertible that fits the customer's true desire. This shows the MVP process where each iteration is used to learn and improve based on feedback.

**How it’s Useful:**

* **Customer-Centric**: MVPs help teams align with customer needs, avoiding the waste of building features the customer doesn't want.
* **Quick Learning**: By testing each iteration, teams get fast feedback and make necessary adjustments early, preventing costly mistakes later.
* **Flexibility**: MVPs allow teams to pivot if the initial assumptions were wrong, ensuring the final product is more successful.

### **Real-Life Applications:**

1. **Software Development**: A startup developing an app might release an MVP with just one core feature (e.g., a login system) to gather user feedback. They could then expand based on what users find valuable, rather than building a full app without any input.
2. **Retail**: A company might create a basic prototype of a new product (e.g., a simple, functional design) to test with customers before producing the final version, saving time and resources.
3. **Automotive Industry**: Car manufacturers may first release limited test models of new car features (such as self-driving technology) to gather feedback before rolling out full-scale production.

By using an MVP, businesses can learn quickly, avoid wasted resources, and deliver a final product that truly meets customer needs.

—----------------------------------------------------------------------------------------------------------------------------

### Video 6: **Test Driven Development**

—------------------------------------------------------------------------------------

**Summary of the Lecture on Test-Driven Development (TDD)**

**Definition and Concept of TDD:**

* Test-Driven Development (TDD) is a software development approach where you write test cases *before* writing the code. These tests drive the design and development of your code.
* TDD ensures that the code's behavior is clear before it is written, focusing on what the code is supposed to do.

**Red, Green, Refactor Workflow:**

* **Red:** Write a test that fails because the code doesn't exist yet.
* **Green:** Write the simplest code to make the test pass.
* **Refactor:** Improve the code while keeping it functional. This iterative process helps in maintaining high code quality.

**Benefits of TDD:**

1. **Higher-quality code**: Writing tests first forces developers to think from the caller's perspective, ensuring that the code meets its intended use.
2. **Saves time**: Although writing tests initially seems time-consuming, it saves significant debugging time later.
3. **Confidence in code**: Test cases provide a safety net, ensuring that code changes don’t break existing functionality.
4. **Easier refactoring**: TDD enables faster refactoring, as failing test cases immediately alert developers if a change breaks the code.

**TDD and DevOps:**

* TDD plays a crucial role in DevOps environments, particularly for creating CI/CD pipelines. Automated tests are essential to ensure that changes are deployed smoothly and without bugs.
* Without automated testing, CI/CD pipelines risk pushing defective code into production.

**Real-life Example:** Imagine a developer creating a banking app. Before writing any code, they write test cases ensuring that account transactions, such as deposits and withdrawals, are handled correctly. This ensures that the core functionalities like balance updates, overdraft protection, etc., work as expected before any code is written. When new features, such as loan processing, are added later, the original tests ensure the new code does not disrupt core functions.

In short, TDD helps you code with confidence, write more reliable software, and maintain high quality in dynamic environments.

### Video 7: **Behavior Driven Development**

—---------------------------------------------------------------------------------------------------

**Summary of the Lecture on Behavior Driven Development (BDD)**

**Definition and Concept of BDD:**

* Behavior Driven Development (BDD) focuses on how the system behaves from an *outside-in* perspective, meaning it considers how users interact with the system.
* It differs from Test-Driven Development (TDD), which tests individual components of the system (inside-out).
* BDD is especially useful for integration testing and ensures that you are building the *right* system that meets user expectations, while TDD ensures you are building it correctly.

**Key Workflow:**

1. **Collaboration:** Developers, testers, and customers collaborate to define the behavior they expect from the system.
2. **Gherkin Language:** Behaviors are documented in Gherkin syntax (Given-When-Then format), which is readable by both technical and non-technical stakeholders.
3. **BDD Tools:** Tools like Cucumber, Behave, and jBehave are used to automate the execution of these behaviors as acceptance tests.

**Benefits of BDD:**

1. **Improves Communication:** BDD creates a shared understanding among team members (developers, testers, product owners, and stakeholders) by using Gherkin, a simple natural language format.
2. **Reduces Misunderstandings:** Since BDD focuses on clear, customer-driven behavior, it ensures the system meets user needs and avoids ambiguity.
3. **Automation:** BDD tools allow you to automate acceptance tests based on the behavior defined in Gherkin syntax, saving time during development.
4. **Higher-quality Code:** BDD leads to more precise guidance on system behavior, improving code quality and reducing maintenance costs.

**Real-life Example:** Consider an online retail store. The BDD scenario for returning an item to stock might be:

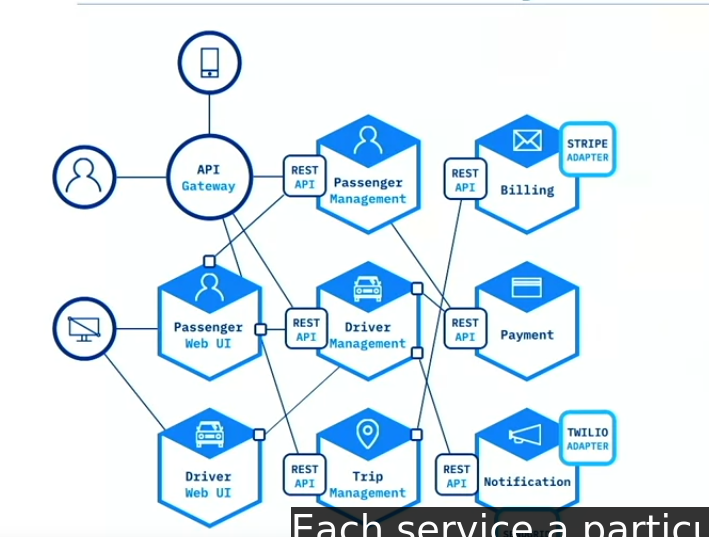
* **Given:** A customer bought a black sweater, and there are 3 black sweaters in stock.
* **When:** The customer returns the black sweater.
* **Then:** The system should update the stock to show 4 black sweaters available. This scenario is clear, understandable, and easy for stakeholders to validate. Using BDD tools, developers can automate testing to ensure the system performs this function correctly, improving quality and reducing misunderstandings.

**Why BDD is Useful:** BDD ensures that the system functions according to customer expectations, improves team collaboration, and speeds up the development process through test automation. It ensures you're building what the user really needs, reducing risks and improving customer satisfaction.

—----------------------------------------------------------------------------------------------------------------------------

### Video 8: **Cloud Native Microservices**

—------------------------------------------------------------------------------------------



### **Summary of Key Points on Cloud Native Microservices and Their Impact on Application Design**

1. **Cloud Native Microservices**:  
   Cloud native architecture consists of small, independent microservices that operate around specific business domains. Each service operates independently, making it easier to scale, manage, and modify without affecting the entire application. This approach is highly beneficial in cloud environments where horizontal scaling is important.
2. **Stateless Microservices**:  
   Microservices are stateless, meaning they do not maintain internal hidden states. Instead, state information is stored in separate databases or persistent object stores. Each service is responsible for its own state, enabling independence and easier scaling. Sharing state across services would turn the architecture into a distributed monolith, which microservices aim to avoid.
3. **Monolithic vs. Microservices Architecture**:  
   In a **monolithic architecture**, all components are tightly coupled, meaning any change or update requires redeploying the entire system. In contrast, **microservices architecture** allows individual services to be deployed, updated, and scaled independently. This leads to better resource utilization and faster development cycles, as different teams can work on services independently without disrupting the entire application.
4. **Resilience and Scaling**:  
   Microservices enable horizontal scaling, where individual services (such as notifications or payment processing) can be scaled independently based on demand. If a service fails, it is simply replaced by another instance, reducing downtime and improving resilience.
5. **REST APIs**:  
   Microservices communicate through lightweight mechanisms like REST APIs, which provide a standard interface for services to interact without knowing each other's internal workings. This decouples the services, allowing developers to change the backend (e.g., from SQL to NoSQL databases) without impacting other services.

### **Real-Life Examples of Microservices**

* **Ride-Sharing Application**:  
  A ride-sharing platform could have separate microservices for handling drivers, payments, trip management, and notifications. Each microservice operates independently, so if the notifications service experiences high traffic, it can be scaled independently without scaling the entire system.
* **E-Commerce Platform**:  
  In a traditional monolithic e-commerce platform, changes to the customer table might require coordination with the order and shipping teams, as they rely on the same database. With a microservices architecture, each service (e.g., customer service, order service, shipping service) would maintain its own database, allowing for faster updates and deployments. If the customer service changes from a SQL to NoSQL database, the order and shipping services wouldn’t be affected because they interact only through REST APIs.

### **Benefits and Usefulness:**

* **Faster Development and Deployment**:  
  With independent services, updates and bug fixes can be applied more frequently, allowing teams to iterate faster.
* **Scalability**:  
  Microservices allow for granular scaling based on individual service demand, reducing costs and improving performance.
* **Resilience**:  
  Failures are isolated within services, preventing system-wide crashes. Failed instances can easily be replaced without affecting other services.

In sum, cloud-native microservices are essential for developing scalable, resilient, and easily maintainable applications in today's cloud environments.

—----------------------------------------------------------------------------------------------------------------------------

### Video 9: **Designing for Failure**

—------------------------------------------------------------------

### **Summary of Key Points on Embracing Failure and Building Resilient Applications**

1. **Embracing Failure**:  
   Failure is inevitable in modern applications, especially those using stateless microservices, which involve numerous interconnected components. Instead of focusing on avoiding failure, developers must design systems to *recover quickly* when failures happen. The key metric shifts from "mean time to failure" to "mean time to recovery." Quick recovery ensures that systems remain functional, even when some components fail temporarily.
2. **Importance of Resilience in Cloud Native Applications**:  
   In a microservices architecture, where services are external and prone to network or service issues, building resilience is critical. Applications need to gracefully handle failures such as slow responses, outages, or throttling (when services restrict usage due to limits). Developers must plan for these eventualities and include logic like retries or caching to mitigate their effects.
3. **Patterns for Building Resilient Applications**: Several design patterns help create robust, failure-resistant applications:
   * **Retry Pattern**:  
     This pattern allows an application to handle transient failures by retrying operations. For example, when a service is slow or unavailable, the application should retry after increasing intervals (exponential backoff) to avoid overwhelming the service.  
     *Real-life example*: An e-commerce website experiencing a temporary failure in its payment gateway retries the transaction several times with increasing delays, rather than overwhelming the gateway with repeated requests.
   * **Circuit Breaker Pattern**:  
     Similar to electrical circuit breakers, this pattern prevents cascading failures by stopping calls to a failing service. Once a failure threshold is reached, the circuit opens, and instead of continuously trying the failed service, it returns an error or an alternative response. After some time, it checks if the service is functional again.  
     *Real-life example*: If a ride-sharing app's notification service fails, the circuit breaker cuts off further attempts to call it, showing a generic message until the service recovers.
   * **Bulkhead Pattern**:  
     This pattern isolates failing components from the rest of the system. Similar to compartments in a ship, bulkheads prevent one service's failure from affecting others. Each service runs in its own thread pool, ensuring that even if one fails, the others remain operational.  
     *Real-life example*: If a database in a travel booking system fails, the bulkhead pattern ensures that the search and booking services continue functioning, while only the payment service is temporarily impacted.
   * **Chaos Engineering**:  
     This involves intentionally causing failures to test how resilient systems are in practice. Netflix’s "Chaos Monkey" randomly shuts down services to ensure their systems can handle such interruptions and recover gracefully.  
     *Real-life example*: Netflix kills random server instances to simulate failures and see if their streaming services remain operational for users.
4. **Resilience as a Development Concern**:  
   Previously, failure recovery was an operational concern, but with cloud-native microservices, it becomes a development responsibility. Developers need to incorporate resilience into the design from the start, ensuring services are capable of quick recovery and degradation without compromising the entire system.

### **Real-Life Examples of How These Concepts Are Useful:**

* **Ride-Sharing App**:  
  A ride-sharing platform can apply the circuit breaker pattern to its payment service. If the service goes down, the circuit breaker ensures that the app doesn’t continuously try to process payments. Instead, it could show users a friendly error message and retry the payment later.
* **Online Retailer**:  
  In an e-commerce platform, the retry pattern helps ensure that when a customer attempts to place an order and the database is temporarily slow, the system retries the request after increasing intervals, preventing overload and ensuring the order eventually goes through.
* **Streaming Service (Netflix)**:  
  Netflix uses chaos engineering to continuously test the resilience of its microservices. By intentionally killing random services, they ensure their system can handle sudden outages without affecting users' streaming experience.

### **Usefulness of These Strategies:**

These patterns and practices are invaluable in building resilient applications that provide a seamless user experience despite inevitable failures. Instead of service outages or poor performance, users experience graceful degradation, allowing the system to maintain core functionalities even when parts of it fail. By planning for failure and incorporating these patterns, applications become robust, scalable, and more reliable, especially in large-scale, cloud-native environments.

—---------------------------------------------------------------------------------------------------------------------------

# **Summary and Highlights**

Congratulations! You have completed this lesson. At this point in the course, you know:

* Social coding is coding as a community and public repositories and pair programming result in higher code quality.
* Working in small batches reduces waste and means quickly delivering something useful to the customer.
* Minimum viable product is as much about delivery as it is about building what the customer really desires.
* Test driven development is writing the test for the code you wish you had, then writing the code to make the test pass. It allows you to develop faster and with more confidence.
* Behavior driven development focuses on the behavior of the system from the outside in. It looks at the system as a consumer of it.
* Behavior driven development improves communication by using an approachable syntax that developers and stakeholders can understand.
* Microservices are built around business capabilities and are independently deployable by fully automated deployment machinery.
* Cloud native architecture enables independently deployable microservices that take advantage of horizontal scaling and result in more resilient services.
* Failure is inevitable, so we design for failure rather than trying to avoid failure.
* It is important to embrace failure and quickly recover when failures happen.

—----------------------------------------------------------------------------------------------------------------------------

### 

### 

### 

### 

### 