**Net from L2 to L3. Assessment session after improvement plan implementation.**

1. What do you work on at the moment? (within project scope) 1
2. Do you communicate directly with the Client? (retro, demo session, PI planning)
3. Have you ever presented results of other team members to the Customer?
4. Some clarification questions regarding info mentioned in the self-presentation (past projects, role)
5. [What is .Net?](#_.NET_is_a)
6. [What does it consist of? What does it include?](#_.NET_consists_of)
7. Do you have experience with .Net Core? What are the main benefits of working with .Net Core?
8. [What is the difference between thread and task in .Net?](#_In_.NET,_threads)
9. [Could you explain in details how you analyzed performance issues? (related to the info in self-presentation)](#_could_you_explain)
10. [What is the difference between REST and SOAP? How would you choose what to use?](#_Great_question!_Let’s)
11. What kind of documentations have you written in your personal life?
12. [What release should not contain?](#_What_software_release)
13. [What is the difference between vertical and horizontal scaling?](#_Vertical_and_horizontal)
14. [What kind of architecture styles do you know?](architecture#_In_programming,_)
15. [What is the difference between the library and the framework?](#_What_is_the)
16. [What is the difference between design patterns and design principles?](#_What_is_the_1)
17. [What are the roles in the Scrum team?](#_What_are_the)
18. [When you’ve played a role of a Scrum master have you learned something specific?](#_When_you’ve_played)
19. [What is the right case to use KANBAN?](#_What_is_the_2)
20. [What are the phases of SDLC?](#_What_are_the_1)
21. Have you ever tried a pair-programming?
22. How should ideal unit test look like? Do you how to write the unit tests?
23. What other tests have you written for your project?
24. Do you know any drawbacks of your current CI pipeline? What could be improved?
25. Continuous delivery VS deployment (what is the main difference?)
26. What was the most important and influential decision you’ve made on your project?
27. Imagine that your Customer has investigated a huge bug. If you were a Team Lead what would be your first steps.
28. Could you describe what did you do as a Mentor?

# [.NET is a free, open-source developer platform supported by Microsoft](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet" \t "_blank)[1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet). It’s a secure, reliable, and high-performance application platform that allows developers to build many types of applications. Here are some key points about .NET:

* [**Languages**: The primary programming language for .NET is C#, which is strongly-typed, type-safe, and has integrated concurrency and automatic memory management1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet).
* **Cross-Platform**: .NET is supported on various operating systems including Android, Apple, Linux, and Windows. [It’s available in several Linux distributions, such as Red Hat Enterprise Linux and Ubuntu1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet).
* **Libraries**: .NET includes a standard set of libraries and APIs that cover a range of functionalities, from collections to networking to machine learning. [NuGet is the package manager for .NET, containing over 300,000 packages1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet).
* [**Application Models**: With .NET, you can build web apps, mobile apps, desktop apps, microservices, cloud services, machine learning applications, games, and Internet of Things (IoT) apps1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet).
* **Community and Open Source**: .NET is under the .NET Foundation and has an active [1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet). community of developers. [It’s open source, meaning developers and companies can contribute to the platform](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet)
* [**Tools**: Developers can build apps with tools like the .NET CLI, Visual Studio, Visual Studio Code, and can configure continuous integration with services like GitHub Actions1](https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet).

Overall, .NET provides a versatile and powerful environment for developers to create diverse applications across different platforms.

# .NET consists of several key components that work together to provide a comprehensive development framework:

* **Common Language Runtime (CLR)**: The CLR is the execution engine for .NET applications. It provides services such as memory management, exception handling, and security.
* **Base Class Library (BCL)**: The BCL is a large class library that provides a range of functionalities, including file I/O, database interaction, XML document manipulation, and more.
* **Frameworks and SDKs**: .NET includes multiple frameworks like **ASP.NET** for web applications, **.NET MAUI** for cross-platform mobile and desktop apps, and **Entity Framework** for data access.
* **Runtime Languages**: While C# is the most popular, .NET supports multiple languages, including F#, Visual Basic, and more.
* **Development Tools**: Tools like **Visual Studio**, **Visual Studio Code**, and the **.NET CLI** help developers build, debug, and deploy .NET applications.
* **NuGet Packages**: NuGet is the package manager for .NET, allowing developers to use third-party libraries and tools in their applications.
* **.NET Standard**: A formal specification of .NET APIs that are intended to be available on all .NET implementations.
* **Cross-Platform Support**: .NET Core is the cross-platform version of .NET that allows applications to run on Windows, macOS, and Linux.

These components together make .NET a versatile platform, enabling developers to build a wide variety of applications, from web to mobile to desktop to cloud-based services.

# In .NET, **threads** and **tasks** are both used for executing code, but they serve different purposes and operate at different levels of abstraction:

* **Thread**:
  + A **Thread** is a lower-level construct representing an individual unit of execution or a single sequence of instructions.
  + It is part of the operating system and is managed by the .NET Framework’s System.Threading namespace.
  + Threads are useful when you need fine-grained control over parallel execution.
  + There is no built-in mechanism for a thread to return a result to the calling code.
* **Task**:
  + A **Task** represents a higher-level abstraction of asynchronous work that needs to be performed.
  + It is part of the Task Parallel Library (TPL) and is managed by the System.Threading.Tasks namespace.
  + Tasks can return results, and they support cancellation through the use of cancellation tokens.
  + They are easier to work with when implementing asynchronous operations using async and await keywords.
  + Tasks typically use the thread pool, which optimizes and reuses threads, reducing the overhead of thread creation.

[Here are some key differences](https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/)[1](https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/):

* **Result**: A task can return a result, while there is no direct mechanism to return the result from a thread.
* **Cancellation**: Task supports cancellation through the use of cancellation tokens, but Thread doesn’t.
* **Parallelism**: A task can have multiple processes happening at the same time, whereas threads can only have one task running at a time.
* **Asynchronous Programming**: Implementing asynchronous programming is easier with tasks using async and await.
* **Thread Pool**: A new Thread() is not dealing with the thread pool thread, whereas a Task does use thread pool threads.

In summary, use a **Thread** for more control and when you need to maintain state across operations. Use a **Task** for easier and more efficient execution of asynchronous operations, especially when you need to return results or manage cancellation.

# Great question! Let’s break down the differences between REST and SOAP:

**REST (Representational State Transfer)**

* **Architecture Style**: REST is an architectural style that uses standard HTTP methods (GET, POST, PUT, DELETE).
* **Data Format**: Typically uses JSON or XML, but can support other formats like HTML and plain text.
* **Stateless**: Each request from a client to server must contain all the information needed to understand and process the request.
* **Caching**: Responses can be cached to improve performance.
* **Flexibility**: More flexible and easier to use with modern web technologies.

**SOAP (Simple Object Access Protocol)**

* **Protocol**: SOAP is a protocol with strict standards.
* **Data Format**: Uses XML exclusively.
* **Stateful or Stateless**: Can be either stateful or stateless.
* **Security**: Built-in security features like WS-Security.
* **Complexity**: More complex due to its strict standards and extensive features.

**Choosing Between REST and SOAP**

* **Use REST if**:
  + You need a lightweight, flexible, and scalable solution.
  + You are working with web applications and services that require quick and easy integration.
  + You prefer using JSON for data interchange.
* **Use SOAP if**:
  + You need built-in security features and transactional reliability.
  + You are dealing with legacy systems that already use SOAP.
  + You require strict standards and formal contracts for your web services.

Когда мы работаем с REST, можно выделить несколько ключевых элементов:

1. **Использование HTTP методов**: REST API использует стандартные HTTP методы, такие как:
   * **GET** для получения данных.
   * **POST** для создания новых ресурсов.
   * **PUT** для обновления существующих ресурсов.
   * **DELETE** для удаления ресурсов.
2. **URI (Uniform Resource Identifier)**: Каждый ресурс в REST API идентифицируется уникальным URI. Например, https://api.example.com/users/123 может указывать на пользователя с ID 123.
3. **Форматы данных**: REST API обычно использует JSON или XML для передачи данных между клиентом и сервером. JSON является наиболее популярным из-за его легкости и читаемости.
4. **Безсостояние**: Каждый запрос к серверу должен содержать всю необходимую информацию для его обработки. Сервер не хранит состояние клиента между запросами.
5. **Кэширование**: Ответы от сервера могут кэшироваться для повышения производительности и уменьшения нагрузки на сервер.
6. **Унифицированный интерфейс**: REST API стремится к унифицированному интерфейсу, что означает, что взаимодействие с ресурсами должно быть предсказуемым и стандартизированным.

Эти элементы помогают определить, что мы работаем с REST API. Если у вас есть конкретные вопросы или примеры, с которыми вы работаете, я буду рад помочь!  
  
  
  
  
  
Когда мы работаем с SOAP (Simple Object Access Protocol), можно выделить несколько ключевых элементов:

1. **Использование XML**: SOAP использует XML для форматирования сообщений. Все запросы и ответы представлены в виде XML-документов.
2. **SOAP Envelope**: Каждое SOAP-сообщение содержит элемент <Envelope>, который является корневым элементом и включает в себя <Header> и <Body>.
   * **Header**: Содержит метаинформацию о сообщении, такую как безопасность и маршрутизация.
   * **Body**: Содержит фактические данные запроса или ответа.
3. **Протоколы передачи**: SOAP может работать поверх различных протоколов передачи, таких как HTTP, SMTP, TCP и другие. Однако, HTTP является наиболее часто используемым.
4. **WS-Security**: SOAP поддерживает расширенные функции безопасности, такие как WS-Security, которые обеспечивают шифрование и аутентификацию сообщений.
5. **Стандарты и спецификации**: SOAP строго следует стандартам и спецификациям, что обеспечивает совместимость между различными системами и платформами.
6. **WSDL (Web Services Description Language)**: SOAP-сервисы часто описываются с помощью WSDL, который определяет, какие операции доступны, какие параметры требуются и какие типы данных используются.

# What software release should not contain?

A software release should avoid containing the following:

1. **Bugs and Errors**: Ensure thorough testing to minimize the presence of bugs and errors.
2. **Incomplete Features**: Only include features that are fully developed and tested.
3. **Sensitive Information**: Avoid embedding any confidential or sensitive data within the software.
4. **Unnecessary Dependencies**: Ensure the software does not rely on unnecessary external libraries or tools that could complicate installation or usage.
5. **Poor Documentation**: Provide clear and comprehensive documentation; avoid releasing software without adequate user guides and technical documentation.
6. **Hardcoded Credentials**: Never include hardcoded usernames, passwords, or API keys.
7. **Unoptimized Code**: Ensure the code is optimized for performance and does not contain inefficient algorithms or processes.
8. **Deprecated Functions**: Avoid using outdated or deprecated functions that may not be supported in future updates.

# Vertical and horizontal scaling are two approaches to improving the performance and capacity of a system, particularly in the context of computing and databases.

**Vertical Scaling (Scaling Up)**

Definition: Adding more power (CPU, RAM, storage) to an existing machine.

**Advantages:**

Simplicity: Easier to implement since it involves upgrading a single server.

Less complexity: No need to modify the application to work across multiple servers.

**Disadvantages:**

Limited by hardware: There’s a physical limit to how much you can upgrade a single machine.

Downtime: Upgrading hardware often requires downtime.

**Horizontal Scaling (Scaling Out)**

Definition: Adding more machines to a system and distributing the load across them.

**Advantages:**

Scalability: Easier to scale out by adding more machines.

Redundancy: Improved fault tolerance since the failure of one machine doesn’t bring down the entire system.

**Disadvantages:**

Complexity: Requires more sophisticated software to distribute tasks and manage multiple machines.

Cost: Can be more expensive due to the need for additional hardware and software.

**Summary**

Vertical Scaling is like upgrading your computer with more powerful components.

Horizontal Scaling is like adding more computers to share the workload.

# In programming, “architecture styles” refer to the overarching structures and patterns used to design software systems. Here are some common ones:

1. **Monolithic Architecture**: A single, unified codebase where all components are interconnected and interdependent. It’s simple to develop but can become unwieldy as the application grows.
2. **Microservices Architecture**: Breaks down an application into smaller, independent services that communicate over a network. This allows for greater flexibility and scalability.
3. **Service-Oriented Architecture (SOA)**: Similar to microservices, but services are typically larger and more coarse-grained. It emphasizes reusability and interoperability.
4. **Event-Driven Architecture**: Components communicate through events, which are messages that indicate a change in state. This style is highly decoupled and can handle complex workflows.
5. **Layered (N-Tier) Architecture**: Divides an application into layers, such as presentation, business logic, and data access. Each layer has a specific responsibility, promoting separation of concerns.
6. **Client-Server Architecture**: Splits the system into two main components: the client, which requests services, and the server, which provides them. This is a foundational model for many networked applications.
7. **Peer-to-Peer (P2P) Architecture**: Each node in the network can act as both a client and a server. This decentralized approach is used in applications like file sharing and blockchain.
8. **Model-View-Controller (MVC) Architecture**: Separates an application into three interconnected components: the model (data), the view (UI), and the controller (business logic). This promotes organized and maintainable code.
9. **Microkernel Architecture**: Consists of a core system (microkernel) that provides minimal functionality, with additional features implemented as plugins. This is common in operating systems and extensible applications.
10. **Serverless Architecture**: Developers write code without managing the underlying infrastructure. Cloud providers handle the server management, scaling, and maintenance.

# What is the difference between the library and the framework?

**Library**

* **Definition**: A library is a collection of pre-written code that developers can use to perform common tasks.
* **Usage**: Developers call functions or methods from the library in their own code.
* **Control**: The developer is in control of the application’s flow. They decide when and how to use the library.
* **Example**: jQuery is a popular JavaScript library that simplifies HTML document traversal and manipulation.

**Framework**

* **Definition**: A framework is a comprehensive platform that provides a foundation and structure for developing applications.
* **Usage**: Developers build their applications within the framework’s structure.
* **Control**: The framework dictates the flow of the application. It often calls the developer’s code at specific points (known as “inversion of control”).
* **Example**: Angular is a framework for building web applications, providing a complete solution including data binding, routing, and more.

In summary, a **library** offers tools that you can use as needed, while a **framework** provides a structured environment that guides the development process.

# What is the difference between design patterns and design principles?

Design patterns and design principles are both crucial concepts in software development, but they serve different purposes and operate at different levels of abstraction. Here’s a breakdown of their differences:

**Design Patterns**

* **Definition**: Design patterns are reusable solutions to common problems in software design. They are like templates that can be applied to solve specific design issues.
* **Usage**: They provide a proven approach to solving recurring problems, making code more flexible, reusable, and maintainable.
* **Examples**:
  + **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
  + **Observer**: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.
  + **Factory Method**: Defines an interface for creating an object but lets subclasses alter the type of objects that will be created.

**Design Principles**

* **Definition**: Design principles are high-level guidelines that help developers make decisions about system design. They are more abstract and provide a foundation for good software design.
* **Usage**: They guide the overall design process, ensuring that the system is robust, scalable, and maintainable.
* **Examples**:
  + **SOLID Principles**:
    - **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
    - **Open/Closed Principle (OCP)**: Software entities should be open for extension but closed for modification.
    - **Liskov Substitution Principle (LSP)**: Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
    - **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use.
    - **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should depend on abstractions.
  + **DRY (Don’t Repeat Yourself)**: Avoid duplication of code by abstracting out common functionality.
  + **KISS (Keep It Simple, Stupid)**: Simplicity should be a key goal in design, and unnecessary complexity should be avoided.

In summary, **design patterns** are specific solutions to common problems, while **design principles** are broader guidelines that influence overall design decisions. Both are essential for creating high-quality software.

# What are the roles in the Scrum team?

In a Scrum team, there are three key roles:

1. **Product Owner**: This person is responsible for maximizing the value of the product by managing the product backlog. They prioritize tasks based on business value and stakeholder needs.
2. **Scrum Master**: The Scrum Master ensures that the team adheres to Scrum practices and principles. They facilitate meetings, remove impediments, and help the team improve their processes.
3. **Developers**: This group includes all the team members who work on delivering the product increment. They are responsible for planning, developing, testing, and delivering the product.

Each role is crucial for the success of the Scrum framework.

# When you’ve played a role of a Scrum master have you learned something specific?

I don’t have personal experiences, but I can share what many Scrum Masters often learn in their roles:

1. **Facilitation Skills**: Effective facilitation of meetings and discussions is crucial. Scrum Masters learn to guide conversations, ensure everyone’s voice is heard, and keep the team focused on their goals.
2. **Conflict Resolution**: Handling conflicts within the team and finding ways to resolve them constructively is a key skill. This helps maintain a healthy team dynamic.
3. **Continuous Improvement**: Scrum Masters often learn the importance of fostering a culture of continuous improvement. They help the team reflect on their processes and find ways to enhance productivity and quality.
4. **Adaptability**: Every team and project is different. Scrum Masters learn to adapt their approach based on the unique needs and challenges of each situation.
5. **Empathy and Support**: Understanding team members’ perspectives and providing support when needed is vital. This helps build trust and a positive working environment.

Is there a specific aspect of the Scrum Master role you’re curious about?

# What is the right case to use KANBAN?

Kanban is a flexible and visual workflow management method that can be particularly effective in several scenarios:

1. **Continuous Delivery**: If your team needs to deliver work continuously rather than in fixed iterations, Kanban is a great fit. It allows for a steady flow of tasks and quick adjustments.
2. **High Variability in Tasks**: When the nature of tasks varies greatly and priorities change frequently, Kanban helps manage this variability by visualizing the workflow and making it easier to adapt.
3. **Maintenance and Support Work**: For teams handling a lot of maintenance, support, or operational tasks, Kanban can help manage and prioritize incoming work efficiently.
4. **Improving Existing Processes**: If your team is looking to improve an existing process rather than implementing a new one, Kanban can help identify bottlenecks and areas for improvement.
5. **Small Teams or Projects**: Kanban works well for small teams or projects where the focus is on managing individual tasks rather than large-scale planning.
6. **Visualizing Work**: If your team benefits from visualizing the workflow to understand the status of tasks and identify issues quickly, Kanban’s visual boards are very effective.

# What are the phases of SDLC?

The Software Development Life Cycle (SDLC) consists of several phases that guide the development process from initial planning to deployment and maintenance. Here are the typical phases:

1. **Planning**: This phase involves defining the project scope, objectives, and feasibility. It includes gathering requirements and creating a project plan.
2. **Requirements Analysis**: In this phase, detailed requirements are gathered from stakeholders to understand what the software needs to achieve. This often involves creating requirement specifications.
3. **Design**: The design phase involves creating the architecture of the software. This includes both high-level design (overall system architecture) and detailed design (specific components and interfaces).
4. **Implementation (Coding)**: During this phase, developers write the actual code based on the design documents. This is where the software is built.
5. **Testing**: Once the software is developed, it undergoes various types of testing (unit testing, integration testing, system testing, etc.) to identify and fix bugs and ensure it meets the requirements.
6. **Deployment**: After successful testing, the software is deployed to the production environment where it will be used by end-users.
7. **Maintenance**: Post-deployment, the software enters the maintenance phase where it is updated and improved based on user feedback and changing requirements. This phase also involves fixing any issues that arise.

Each phase is crucial for ensuring the development of high-quality software. Do you have any specific questions about any of these phases?

# could you explain in details how you analyzed performance issues?

Absolutely! Analyzing performance issues in .NET applications involves several steps and tools to identify and resolve bottlenecks. Here’s a detailed approach:

**1. Identify Symptoms**

Start by identifying the symptoms of the performance issue. This could be slow response times, high CPU usage, memory leaks, or application crashes.

**2. Use Profiling Tools**

Profiling tools help you understand where the application spends most of its time and resources. Some popular tools include:

* [**Visual Studio Profiler**: Integrated into Visual Studio, it provides detailed insights into CPU usage, memory allocation, and I/O operations](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0)[1](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0).
* [**dotTrace**: A performance profiler for .NET applications that helps identify slow methods and heavy resource usage](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0)[2](https://www.youtube.com/watch?v=hgXadZgddO0).
* [**PerfView**: A tool for analyzing CPU usage, memory, and GC behavior](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0)[1](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0).

**3. Collect Data**

Run your application under typical load conditions and use the profiling tools to collect data. This involves:

* **CPU Sampling**: To identify which methods consume the most CPU time.
* **Memory Profiling**: To detect memory leaks and excessive memory usage.
* **I/O Profiling**: To find slow database queries or file operations.

**4. Analyze the Data**

Examine the collected data to pinpoint the performance bottlenecks:

* **Hotspots**: Identify methods or operations that take the most time.
* **Call Trees**: Understand the call hierarchy and how time is distributed among methods.
* **Garbage Collection**: Check for frequent or long garbage collection pauses.

**5. Optimize Code**

Based on the analysis, optimize the code:

* **Refactor Inefficient Code**: Rewrite or optimize methods that are identified as bottlenecks.
* **Improve Algorithms**: Use more efficient algorithms or data structures.
* **Reduce I/O Operations**: Optimize database queries and reduce file I/O.

**6. Monitor in Production**

Use monitoring tools to keep an eye on the application’s performance in production:

* [**Application Insights**: Provides real-time monitoring and diagnostics1](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0).
* [**Performance Counters**: Track CPU, memory, and other metrics](https://learn.microsoft.com/en-us/aspnet/core/performance/diagnostic-tools?view=aspnetcore-8.0)[3](https://michaelscodingspot.com/performance-tools-dotnet/).

**7. Iterate**

Performance tuning is an iterative process. Continuously monitor, profile, and optimize your application to maintain optimal performance.

Would you like more details on any specific tool or step?