Task 1: how to increase security level in functional programming:

1. Immutability: Embrace immutability wherever possible. Immutable data structures and variables reduce the risk of unintended side effects and make it harder for attackers to manipulate the program state. (Immutability means an object that doesn't allow changes in its value).
2. Pure Functions: Write pure functions that don't rely on external state or side effects. Pure functions are easier to reason about and test, reducing the likelihood of bugs and security vulnerabilities.
3. Input Validation: Sanitize and validate all input data thoroughly. Untrusted input can lead to vulnerabilities like injection attacks, so ensure that the data conforms to expected formats and limits.
4. Avoid Recursion Vulnerabilities: Be mindful of recursive function calls that could lead to stack overflow vulnerabilities. Some functional languages optimize tail-recursive functions to prevent this issue.
5. Avoid Exposing Sensitive Data: Keep sensitive information like passwords, access tokens, and private keys out of the codebase. Use environment variables or configuration files to manage such data securely.
6. Use Standard Libraries: Rely on well-tested and maintained standard libraries for tasks like cryptography, hashing, and input parsing. These libraries are more likely to be secure and free of common vulnerabilities.
7. Regular Code Reviews: Conduct thorough code reviews to catch potential security flaws before they make it into production. A fresh pair of eyes can identify issues that the original developer might have overlooked.
8. Security Testing: Perform security testing, including penetration testing and fuzz testing, to identify vulnerabilities and weaknesses in your functional code.
9. Stay Updated: Keep your functional programming language and libraries up-to-date with the latest security patches and updates.
10. Employ HTTPS: If your functional program interacts with external services or APIs, ensure that all communications are encrypted using HTTPS.

Task 2: Declarative programming and Functional programming:

Declarative Programming:

Declarative programming is a paradigm for writing computer programs that places an emphasis on explaining the logic and rules of a computation rather than outlining the precise steps needed to carry it out. Developers announce what they intend to happen rather than providing specific instructions, and the underlying system or language takes care of the implementation specifics.

The main key characteristics of Declarative programming are: focus on “What” rather than “How”, Abstraction, and readability.

SQL (Structured Query Language), which is used for database queries, and CSS (Cascading Style Sheets), which is used to describe the structure and display of web pages, are two common instances of declarative programming.

Functional Programming

Functional programming is a subset of declarative programming that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. In functional programming, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned as results.

The main key characteristics of Functional programming are: pure function, immutability, and recursion.

Haskell, Lisp, and more contemporary languages like Scala, Clojure, and portions of JavaScript are examples of functional programming languages.

Because they concentrate on what must be calculated (declarations of functions and data transformations) rather than on how they should be computed (control flow details), functional programming languages can be declarative. As there are various ways to describe "what" without necessarily requiring functional programming ideas, not all declarative languages are functional.

Task 3: what are solid principles:

1. Single Responsibility Principle (SRP): According to this principle, a class should only have one responsibility and one valid justification for change. To make it simpler to comprehend, maintain, and modify code, a class should only include one functionality or behavior. Changes to one of a class's several functions may unintentionally influence the others, creating a tightly connected and vulnerable design.
2. Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for expansion but closed for modification, according to the Open/Closed Principle (OCP). In other words, a module's behavior should be extendable without requiring changes to the code that makes up the module. Usually, inheritance, interfaces, or abstractions are used to accomplish this. This idea makes it possible to introduce new features without changing the old code, which lowers the chance of adding defects.
3. Liskov Substitution Principle (LSP): Objects of a superclass should be interchangeable with objects of its subclasses without impacting the program's correctness, according to the Liskov Substitution Principle (LSP). To put it another way, a derived class should be able to replace its base class without altering the system's desired attributes. By doing this, it is made possible for every subclass of the base class that depends on it to function correctly.
4. Interface Segregation Principle (ISP): The Interface Segregation Principle (ISP) contends that clients shouldn't be made dependent on interfaces they do not use. In other words, having several smaller, more specialized interfaces is preferable to having one huge, all-purpose interface. This rule promotes a more unified design by preventing a class from being overburdened with methods that it doesn't require.
5. Dependency Inversion Principle (DIP): High-level modules shouldn't be dependent on low-level modules, according to the Dependency Inversion Principle (DIP). They ought to both rely on abstractions. Details should not depend on abstractions, and abstractions should not depend on details. In order to specify the contract between components, this principle promotes the usage of interfaces or abstract classes, allowing for flexible and interchangeable implementations.

How to avoid problems that require solid principles:

Problems can be avoided by the following strategies: plan and design ahead, modularize the code, use abstractions, reviewing code regularly, unit testing, refactoring when necessary, and adopting design patterns.

Task 4: how to achieve scalability using functional programming:

1. Immutable Data Structures: Favor immutable data structures and avoid shared mutable state. Immutable data allows for easy sharing between threads or processes, reducing the need for synchronization and minimizing the risk of race conditions.
2. Concurrency and Parallelism: Functional programming encourages the use of pure functions, which are inherently thread-safe and can be executed in parallel without side effects. Leveraging concurrent and parallel programming techniques can lead to significant performance improvements.
3. Functional Composition: Emphasize function composition to build complex functionalities from smaller, composable functions. This makes it easier to reason about your code and optimize specific parts independently.
4. Lazy Evaluation: Some functional programming languages support lazy evaluation, where computations are delayed until their results are actually needed. Lazy evaluation can improve performance by avoiding unnecessary computations.
5. Memoization: Consider using memoization to cache the results of expensive function calls. This technique is particularly useful when dealing with recursive or computationally intensive operations.
6. MapReduce: Utilize functional programming's map and reduce operations for distributed data processing. These operations can be easily parallelized and scaled across a cluster of machines, making them ideal for big data processing.
7. Avoid Global State: Minimize the use of global state and shared mutable data. Global state can introduce hidden dependencies and make it harder to scale your application.
8. Functional Data Pipelines: Implement data pipelines using functional programming constructs. Data pipelines can be scaled horizontally by distributing tasks across multiple nodes or workers.
9. Distributed Systems: Consider using functional programming to build distributed systems. Functional programming's emphasis on immutability and pure functions can make it easier to reason about distributed computations and data flows.
10. Use Message Passing: In distributed systems, prefer message passing over shared memory for communication between different components. Message passing helps in achieving loose coupling and can simplify the scaling process.
11. Microservices Architecture: Break your application into smaller, independent microservices that can be deployed and scaled independently. Each microservice can be implemented using functional programming principles.
12. Horizontal Scaling: Scale your application horizontally by adding more instances of functional components as the demand increases. Functional programming's statelessness and immutability facilitate horizontal scaling.
13. Optimize Tail Recursion: In languages that support tail call optimization, use recursion to build expressive algorithms without the risk of stack overflow errors.

Task 5: search about code Injection:

Different types of code injection:

1. SQL Injection (SQLi): In SQL injection attacks, malicious SQL statements are inserted into input fields that are used to construct SQL queries. If the application does not properly validate and sanitize user input, attackers can manipulate the SQL query and gain unauthorized access to the database or perform unauthorized operations.
2. Cross-Site Scripting (XSS): XSS attacks involve injecting malicious scripts (usually JavaScript) into web pages viewed by other users. This occurs when user-provided data is not properly sanitized before being rendered in the HTML response. As a result, attackers can steal user session information, redirect users to malicious websites, or perform other malicious actions.
3. OS Command Injection: In OS command injection attacks, untrusted input is directly used in operating system commands without proper validation. Attackers can execute arbitrary commands on the system, leading to unauthorized access, data loss, or system compromise.
4. LDAP Injection: LDAP injection attacks occur when untrusted data is used to construct LDAP (Lightweight Directory Access Protocol) queries. Attackers can manipulate the query to access sensitive data or perform unauthorized actions within the directory service.
5. XPath Injection: XPath injection attacks exploit vulnerabilities in applications that use XPath queries to extract data from XML documents. By injecting malicious data, attackers can modify the XPath query to access unauthorized data.
6. Code Injection in Compiled Languages: In compiled languages, code injection can occur if user input is directly used in the source code without proper validation. Attackers can manipulate the code logic, leading to unexpected behavior or security breaches.

To prevent code injection attacks, developers should follow secure coding practices:

- Use parameterized queries or prepared statements to prevent SQL injection.
- Employ output encoding or validation to prevent XSS attacks.
- Use least-privileged access rights and avoid executing system commands based on untrusted input.
- Implement proper input validation and sanitization for all user-provided data.
- Use context-specific escaping mechanisms for different output contexts (HTML, JSON, etc.).
- Keep software and libraries up-to-date to avoid known vulnerabilities.
- Regularly perform security testing and code reviews to identify and fix potential vulnerabilities.