**Notes**  
**IntegrationStrategies**  
If the volume of backend data grows substantially after finishing the integration, it can lead to various challenges in the software integration strategy. Firstly, the existing infrastructure and systems may not be able to handle the increased data load efficiently, causing performance issues such as slow transfer speeds or system crashes. Additionally, the integration may not have been designed to scale effectively to accommodate large volumes of data, resulting in bottlenecks and resource constraints. This could lead to data loss, corruption, or inaccuracies during file transfers.

Moreover, the FTP file transfer method, while effective for transferring files up to 1 GB in size, may struggle to handle larger files or increased data traffic. The seamless transfer of files and strong firewall settings may become overwhelmed by the sheer volume of data, leading to security vulnerabilities or breaches.

In order to address this issue, TekAlli would need to reassess the software integration strategy and make necessary adjustments to accommodate the increased data volume. This may involve upgrading hardware, optimizing software algorithms, implementing data compression techniques, or leveraging cloud-based solutions for scalability. Additionally, thorough testing and performance monitoring would be crucial to ensure the continued reliability and efficiency of the file transfer solution as the backend data volume grows.

If users ask for modifications after finishing the integration, it can impact the software integration strategy in several ways. Firstly, any modifications or changes to the system may require additional development work, potentially disrupting the existing integration and introducing new bugs or errors. This could result in downtime or service interruptions, impacting the user experience and productivity.

Furthermore, accommodating user requests for modifications may necessitate revisiting the initial design and architecture of the integration. This could lead to scope creep, where the project expands beyond its original parameters, resulting in delays and cost overruns. Additionally, if the modifications are not properly planned and implemented, they could introduce compatibility issues or conflicts with existing systems or functionalities.

To address this challenge, TekAlli would need to adopt an agile approach to software development, allowing for flexibility and responsiveness to user feedback and change requests. This may involve establishing clear communication channels with users to prioritize and manage modifications effectively. Additionally, rigorous testing and quality assurance processes would be essential to ensure that any modifications do not compromise the integrity or stability of the integrated system. Overall, proactive and collaborative engagement with users would be key to successfully managing modifications and maintaining the effectiveness of the software integration strategy.  
  
  
  
**Data File Architecture**

-Just assume we are receiving a data file which consists of,

Invoice information

Payment information related to the invoices already delivered.

Notes regarding already delivered invoices.

And Credit Notes (Invoice Cancellations)

Just forget about what information we need to deal with for the moment and just think about what type of data we need to import into the system, which are mentioned below

• Invoices

• Payments

• Notes

• Credit Notes

Those are the four types of data that come with the data files. All we need to do is transform the data file content into a data object with the above-mentioned properties. So, we have to make sure we read the data properly and have them stored in the data objects in the end.

Pipes and Filters diagram? Challenges advantages

To implement an Architectural Pattern Solution using Pipes and Filters for importing data into the system, we can break down the process into a series of stages (filters) connected by pipes, where each stage performs a specific transformation or processing on the data. Here's how we can implement it:  
**Pipes and Filter architecture**

1. Input Source (Pipe):

• The input source is responsible for reading the data file and providing it as input to the pipeline.

• It could be a file reader component that reads the data file line by line or in batches.

2. Data Parsing (Filter):

• The data parsing filter processes the raw data received from the input source and parses it into separate records based on the data types (invoices, payments, notes, credit notes).

• It identifies and extracts relevant information for each record type.

3. Data Transformation (Filters):

• Separate filters are applied for each record type to perform any necessary data transformations or validations.

• For example, for invoices, validation checks could be performed for invoice amounts, dates, and other attributes.

• Similarly, for payments, validations could be performed against invoice IDs, payment amounts, etc.

4. Data Storage (Filters):

• Once the data has been parsed and transformed, separate filters are responsible for storing the data into the appropriate data objects (e.g., database tables, data structures).

• Each record type may have its own storage filter to handle the specific requirements of that data type.

Challenges involved in this solution:

• Ensuring data consistency and integrity across different stages of the pipeline.

• Handling errors and exceptions gracefully to prevent data loss or corruption.

• Managing the complexity of the pipeline as more stages and transformations are added.

• Dealing with performance issues, especially when processing large volumes of data.

Advantages of the Pipes and Filters pattern for the given scenario:

• Modularity: The solution is modular and can be easily extended or modified by adding or removing filters as needed.

• Reusability: Filters can be reused across different pipelines or scenarios, promoting code reuse and maintainability.

• Scalability: The pipeline architecture allows for parallel processing of data, enabling scalability to handle large datasets efficiently.

• Separation of Concerns: Each filter is responsible for a specific task or concern, leading to clearer and more maintainable code.

• Testability: Filters can be individually unit tested, allowing for easier testing and validation of each processing step.

* How will you implement Service Oriented Architecture for developing the software benefit of using it

**Service Oriented Architecture**

To implement a gaming software using Service-Oriented Architecture (SOA), we can break down the system into a set of loosely coupled, independently deployable services that communicate with each other via standardized protocols. Here's how we can implement SOA for developing the gaming software:

1. Service Identification:

• Identify the functional components of the gaming software that can be encapsulated as services.

• For example, services could include user authentication, game logic, player matchmaking, leaderboard management, in-game purchases, etc.

2. Service Design:

• Design each service to encapsulate a specific piece of functionality, with clear boundaries and well-defined interfaces.

• Define service contracts specifying the inputs, outputs, and behavior of each service.

• Use industry-standard protocols such as RESTful APIs or messaging protocols like MQTT for communication between services.

3. Service Implementation:

• Develop each service as an independent module, using appropriate programming languages and frameworks.

• Implement robust error handling and fault tolerance mechanisms to ensure service reliability and availability.

• Use containerization technologies like Docker to package and deploy services in a scalable and portable manner.

4. Service Orchestration:

• Define workflows or orchestration logic to coordinate interactions between different services to fulfill user requests or execute game actions.

• Use workflow engines or message brokers to manage service interactions and ensure reliable message delivery.

5. Service Discovery and Invocation:

• Implement service discovery mechanisms to dynamically locate and invoke services at runtime.

• Utilize service registries or service meshes to manage service endpoints and facilitate service-to-service communication.

Benefits of using Service-Oriented Architecture for gaming software development:

1. Modularity and Reusability:

• Services can be developed, deployed, and updated independently, promoting code reuse and modular design.

• Developers can leverage existing services to accelerate development and reduce time-to-market for new features or games.

2. Scalability:

• SOA allows for horizontal scaling by deploying multiple instances of services to handle increasing user loads or demand.

• Services can be scaled independently based on usage patterns or performance requirements.

3. Flexibility and Agility:

• SOA enables flexibility in adapting to changing business requirements or gaming trends.

• New services can be added or existing services modified without disrupting the entire system, fostering agility and innovation.

4. Interoperability:

• Services communicate via standardized interfaces and protocols, enabling interoperability across different platforms, devices, and gaming ecosystems.

• This facilitates integration with third-party services, APIs, or gaming platforms, enhancing the overall gaming experience.

5. Fault Isolation and Resilience:

• Services operate independently, reducing the impact of failures or faults in one service on the rest of the system.

• Fault tolerance mechanisms ensure service availability and reliability, enhancing system resilience and uptime.

a) A window management system is a type of interactive user interface that enables users to work with multiple separate applications at the same time. This is achieved using a desktop metaphor in which each process is associated with a graphical window. A window management system provides the functionality to create and manipulate the display of multiple processes. A window management system includes different important components such as

• Input manager for I/O controlling,

• Process manager for managing application processes,

• Screen manager for maintaining the integrity of the screen and

• Window manager for managing the windows that are related to the application processes.

three architectural patterns suitable for the above scenario. Justify your selection in detail for each of the architectural pattern. Also mention the challenges for implementing the selected architectural pattern.

Multiple business processes in an organization require the user authentication functionality.

implement Service Oriented Architecture to implement the above mentioned scenario? What is the benefit of using it?

**MVC Architecture**

a) Architectural Patterns Suitable for Window Management System:

1. Model-View-Controller (MVC) Pattern:

• Justification:

• MVC separates the application into three interconnected components: Model (data and logic), View (user interface), and Controller (user input processing).

• Input Manager, Process Manager, Screen Manager, and Window Manager can be represented as Controllers, which handle user input, manage application processes, maintain screen integrity, and manage window display respectively.

• The actual application processes and data are represented as Models, which are managed and manipulated by the Controllers.

• Views represent the graphical windows associated with each process, displaying the application's user interface.

• Challenges:

• Ensuring proper communication and synchronization between the Model, View, and Controller components.

• Managing the complexity of interactions between multiple Controllers and Views in a large-scale window management system.

2. **Client-Server Architecture**:

• Justification:

• Client-Server architecture divides the system into client (user interface) and server (back-end services) components, communicating over a network.

• Input Manager, Process Manager, Screen Manager, and Window Manager can be implemented as server-side components, providing services to multiple client applications.

• Each graphical window (View) corresponds to a client application, interacting with the server-side components to manage application processes and display windows.

• Challenges:

• Designing and implementing robust communication protocols and APIs for client-server interactions.

• Ensuring scalability and performance of server-side components to handle multiple client requests simultaneously.

3. **Microservices Architecture**:

• Justification:

• Microservices architecture decomposes the system into small, independently deployable services, each responsible for a specific function or feature.

• Input Manager, Process Manager, Screen Manager, and Window Manager can be implemented as separate microservices, each focusing on a specific aspect of window management.

• Each microservice can be developed, deployed, and scaled independently, allowing for flexibility and modularity in the window management system.

• Challenges:

• Managing the complexity of interactions and dependencies between microservices.

• Ensuring consistency and coordination between distributed components in a distributed system.

b) Implementing Service-Oriented Architecture (SOA) for User Authentication:

• Service Identification: Identify the functional components related to user authentication, such as authentication service, user management service, and session management service.

• Service Design: Design each service with well-defined interfaces for user authentication operations like login, logout, user registration, password reset, etc.

• Service Implementation: Implement each service as an independent module using appropriate programming languages and frameworks, ensuring scalability, security, and reliability.

• Service Orchestration: Define workflows or orchestration logic to coordinate interactions between different services to fulfill user authentication requests.

• Service Invocation: Implement client applications to invoke the authentication services using standardized protocols like RESTful APIs or SOAP, ensuring interoperability and communication reliability.

Benefits of Using SOA:

• Modularity and Reusability: Services can be reused across different business processes and applications, promoting code reuse and reducing development time.

• Scalability: SOA allows for horizontal scaling by deploying multiple instances of services to handle increasing user loads or demand.

• Flexibility and Agility: SOA enables flexibility in adapting to changing business requirements or user needs, allowing for easy modification or addition of authentication services.

• Interoperability: Services communicate via standardized interfaces and protocols, enabling interoperability across different platforms, devices, and applications.

• Security: SOA allows for centralized management and enforcement of security policies and access controls, ensuring secure authentication and user management functionalities.

b)

1. A weather forecasting system needs to process a large volume of data very quickly. The data to be processed include wind speed and direction, temperature, humidity. These data are collected from 10 million locations every 2 minutes. The processing of these data to predict the weather consists of 2 steps: Step 1: Calculate the trend of each parameter during the last 72 hours. Step 2: Correlate the trends to predict the weather for the next 3 days.

patterns can be best suited for the above-mentioned scenario? selection of the pattern with the challenges that may be faced for your selected pattern.

* 3 different architectures using Layered, MVC and Broker for Gaming software and compare them regarding fulfilment of quality attributes.

The game must support following User operations:

• Game can be played on any device.

• Any number of users are possible to play.

• User interface must be good.

• All payers should score marks and highest score to be conveyed to other players.

Winner of the game will be informed and will be rewarded.

a) Architectural Patterns for Weather Forecasting System:

1. Pipeline Pattern:

• Justification:

• The Pipeline pattern is suitable for processing large volumes of data in a sequential manner, with each step of the pipeline performing a specific task.

• Step 1 (calculating trends) and Step 2 (predicting weather) can be represented as separate stages in the pipeline.

• Data collected from 10 million locations every 2 minutes can be efficiently processed by dividing the workload into smaller batches and processing them sequentially through the pipeline.

• Challenges:

• Ensuring scalability and performance of the pipeline to handle the high volume of data within the required time constraints.

• Implementing fault tolerance mechanisms to handle errors or failures in data processing stages without compromising the overall system reliability.

2. Event-Driven Architecture (EDA):

• Justification:

• EDA is well-suited for real-time data processing scenarios where events trigger actions or computations.

• Each data collection event triggers the processing of trends and weather prediction.

• As new data is collected every 2 minutes, the system can continuously process incoming data streams in real-time to provide up-to-date weather forecasts.

• Challenges:

• Designing an efficient event processing system capable of handling the high volume and velocity of incoming data streams.

• Ensuring low latency and high throughput in event processing to maintain real-time responsiveness of the system.

b) Architectures for Gaming Software:

1. Layered Architecture:

• Fulfillment of Quality Attributes:

• Scalability: Layered architecture allows for scalability by separating concerns into layers, making it easier to scale individual layers independently.

• Maintainability: Clear separation of concerns in layers promotes maintainability, as changes to one layer do not affect others.

• User Interface: Presentation layer focuses on user interface design, ensuring a good user experience.

• Comparison: While layered architecture provides good separation of concerns, it may suffer from communication overhead between layers, affecting performance.

2. Model-View-Controller (MVC) Architecture:

• Fulfillment of Quality Attributes:

• Scalability: MVC separates the application into three interconnected components, allowing for scalability by modularizing the system.

• User Interface: View component focuses on user interface design, ensuring a good user experience.

• Communication: Controller acts as an intermediary between Model and View, facilitating communication.

• Comparison: MVC provides good separation of concerns and promotes maintainability, but may suffer from complexity in handling user interactions and state management.

3. Broker Architecture:

• Fulfillment of Quality Attributes:

• Scalability: Broker architecture allows for scalability by decoupling components and facilitating asynchronous communication.

• Communication: Centralized broker facilitates communication between components, enhancing interoperability.

• Flexibility: Allows for dynamic addition or removal of components, promoting flexibility.

• Comparison: While broker architecture provides loose coupling and flexibility, it may introduce single points of failure and increased complexity in managing the broker component.  
  
  
  
--

For parallel -   
1- ((1-R1)(1-R2))  
For Series   
R1 \*R2  
Total benefit /(Cost\*Weight)  
BCR = Σ PV of Benefits (cash inflows) / Σ PV of costs (cash outflows)

**Casandradatabase**

Cassandra is a NoSQL database itself, belonging to the wide category of distributed NoSQL databases. NoSQL databases are chosen for Cassandra-like systems due to their ability to handle large amounts of distributed data and provide scalability, flexibility, and high performance. Some NoSQL databases that are commonly used in conjunction with Cassandra include:

1. **MongoDB:**

MongoDB is a document-oriented NoSQL database that is known for its flexibility and scalability. It stores data in a flexible, JSON-like format, making it suitable for a variety of data types.

2. **Apache CouchDB:**

CouchDB is a distributed NoSQL database that uses a document-oriented approach. It's designed for ease of use, high performance, and fault tolerance.

3. **Apache HBase:**

HBase is a distributed, scalable, and consistent NoSQL database that is built on top of the Hadoop Distributed File System (HDFS). It is suitable for real-time read and write access to large datasets.

4. **Hadoop:**

While Hadoop is primarily known as a distributed storage and processing framework, it includes components like HBase for NoSQL capabilities. Hadoop is often used in conjunction with Cassandra for comprehensive big data solutions.

These databases are chosen based on the specific requirements of the application, such as the nature of the data, scalability needs, and the desired data model.  
  
**NOSQL Architecture**  
The architecture of a NoSQL database system depends on various factors, including the specific database chosen and the requirements of the application. However, a generic architecture for a NoSQL database system might include the following components:

1. **Client Application:**

 The client application interacts with the NoSQL database system to perform operations such as read, write, update, and delete.

2. **Load Balancer:**

 In a distributed environment, a load balancer can distribute incoming requests across multiple nodes to ensure even distribution of the workload.

3. **Node/Server:**

 Nodes or servers host the NoSQL database instances. Each node is responsible for managing a portion of the data.

4. **Database Cluster:**

 A collection of interconnected nodes forms a database cluster. The cluster ensures high availability, fault tolerance, and scalability.

5. **Data Storage:**

 The actual data storage component stores and manages the data. In Cassandra, for example, this includes keyspaces, column families, and data files.

6. **Query Processor:**

 The query processor interprets and executes queries from client applications. It may optimize queries for performance and distribute them across the nodes.

7. **Replication:**

 Replication mechanisms ensure data durability and fault tolerance by creating copies of data on multiple nodes.

8. **Partitioning:**

 Partitioning is essential for distributing data across nodes. It allows the database to scale horizontally by adding more nodes.

9. **Indexes:**

 Depending on the NoSQL database, indexes may be used to optimize query performance.

10. **Security and Authentication:**

 Components for ensuring data security, access control, and user authentication.

The specific architecture can vary based on the chosen NoSQL database and the unique requirements of the application.  
  
**Blockchain**  
  
While both blockchains and traditional databases store and manage data, they serve different purposes and have distinct characteristics:

1. **Immutability and Consensus:**

 Blockchains are designed to provide a tamper-resistant and immutable ledger through the use of consensus mechanisms like proof-of-work or proof-of-stake. Every block in a blockchain is linked cryptographically to the previous one, forming a chain. Traditional databases, on the other hand, do not inherently have these features, and data modifications are subject to the authority of the database administrator.

2. **Decentralization:**

 Blockchains are typically decentralized, with no single point of control. Decentralization ensures that no single entity has absolute control over the entire system. Traditional databases may be centralized or distributed but are often controlled by a central authority.

3. **Smart Contracts and Programmability:**

 Many blockchains, like Ethereum, support smart contracts, which are self-executing contracts with the terms of the agreement directly written into code. This programmability is not a standard feature in traditional databases.

4. **Consistency Model:**

 Blockchains often use a consensus mechanism to achieve a distributed and consistent state across nodes. Traditional databases may use ACID properties (Atomicity, Consistency, Isolation, Durability) for consistency but might not operate in a distributed or decentralized manner.

In summary, while blockchains can be considered a form of a database due to their ability to store and manage data, they are designed with specific features like immutability, decentralization, and consensus that go beyond the capabilities of traditional databases. The reverse is not true because not every traditional database incorporates these blockchain-specific features.  
  
  
**Pipe Filter – Integrity of message – Hashing Filter**   
**Justification:**

 **Hashing Algorithm:**

 A Hashing Filter applies a hashing algorithm to the message data. The algorithm generates a fixed-size hash value (digest) that uniquely represents the content of the message.

 **Verification Mechanism:**

 The hash value is then securely transmitted or stored along with the message. Upon receiving the message, the system recalculates the hash value using the same algorithm.

 **Comparing Hash Values:**

 The recalculated hash value is compared with the original hash value. If they match, it ensures that the message has not been altered during transmission or storage. If there's any modification, the hash values won't match, indicating potential tampering.

 **Preventing Eavesdropping:**

 While a Hashing Filter itself does not encrypt the message, it provides a means to detect unauthorized changes. If eavesdroppers attempt to modify the message, the hash value won't match, and the system can reject the tampered message.   
  
**Pipe filter – Security of data – Encryption filter**

**Encryption Algorithm:**

 An Encryption Filter applies an encryption algorithm to the message data, transforming it into a secure and unreadable format.

 **Data Confidentiality:**

 Encryption ensures that even if an unauthorized entity intercepts the message, they cannot understand its content without the decryption key. This protects the confidentiality of sensitive data.

 **Preventing Unauthorized Access:**

 Only authorized parties with the correct decryption key can access and interpret the original message. This prevents unauthorized access and protects the data from being compromised.

 **Secure Transmission:**

 When the encrypted message is transmitted, it remains secure even if intercepted by eavesdroppers. Without the decryption key, the intercepted data is essentially meaningless.

**to check all incoming messages to provide the security-** Authentication and Authorization Filter

**User Authentication:**

 An Authentication and Authorization Filter verifies the identity of the sender or source of the incoming message. It ensures that the message is from a legitimate and authorized entity.

 **Access Control:**

 Authorization mechanisms within the filter determine whether the sender has the right permissions to perform the requested action. This helps prevent unauthorized access to sensitive operations or data.

 **Integrity Check:**

 In addition to user authentication, the filter may perform integrity checks on the incoming message to ensure that it has not been tampered with during transmission.

 **Logging and Auditing:**

 The filter can log information about incoming messages, providing an audit trail. This is crucial for monitoring system activity and detecting potential security breaches.

By incorporating an Authentication and Authorization Filter, the safety-handling system can maintain control over who can access and modify data, enhancing the overall security of the system.  
  
  
Score = total benefit / (Cost\*weight)