TestpassportQ&A

Bessere Qualität , bessere Dienstleistungen!

We offer free update service for one year

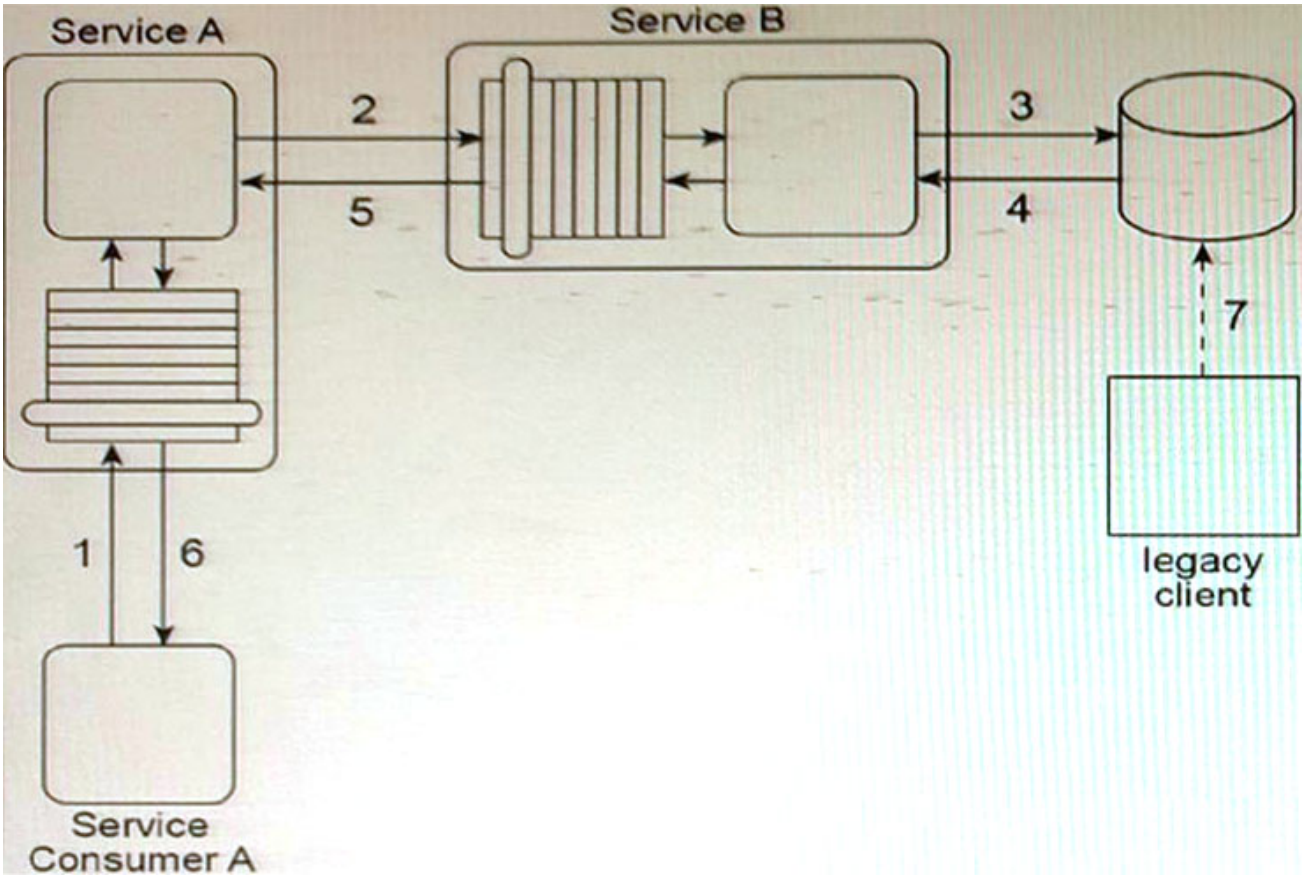
Http://www.testpassport.ch

Exam : **S90.08B**

Title : SOA Design & Architecture Lab with Services & Microservices

Version : DEMO

1.Service A is an entity service that provides a Get capability which returns a data value that is frequently changed.



Service Consumer A invokes Service A in order to request this data value (1). For Service A to carry out this request, it must invoke Service B (2), a utility service that interacts (3, 4) with the database in which the data value is stored. Regardless of whether the data value changed, Service B returns the latest value to Service A (5), and Service A returns the latest value to Service Consumer A (6).

The data value is changed when the legacy client program updates the database (7). When this change will occur is not predictable. Note also that Service A and Service B are not always available at the same time.

Any time the data value changes, Service Consumer A needs to receive It as soon as possible. Therefore, Service Consumer A initiates the message exchange shown In the figure several times a day. When it receives the same data value as before, the response from Service A Is ignored. When Service A provides an updated data value, Service Consumer A can process it to carry out its task.

The current service composition architecture is using up too many resources due to the repeated invocation of Service A by Service Consumer A and the resulting message exchanges that occur with each invocation.

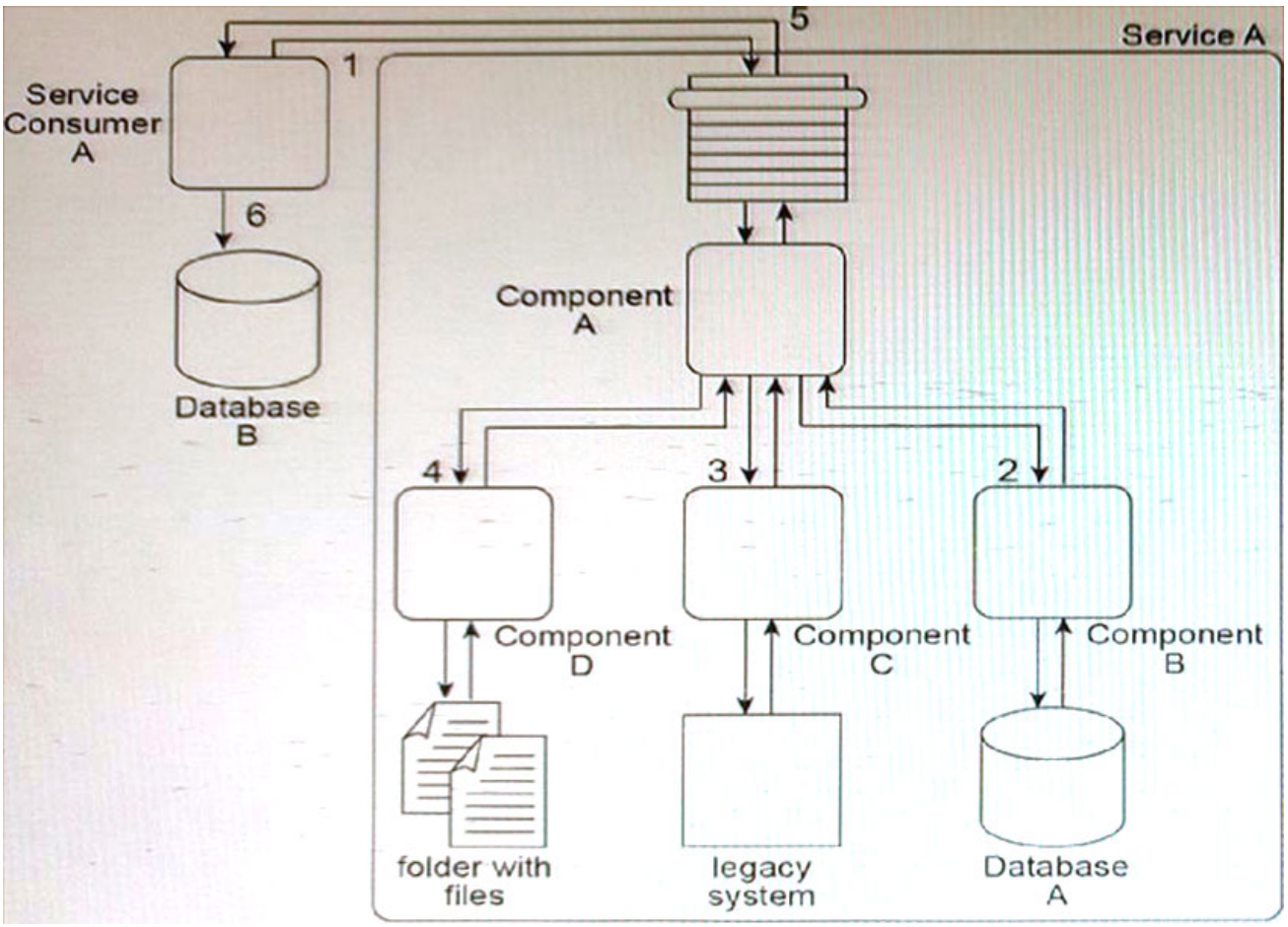
What steps can be taken to solve this problem?

A. The Event-Driven Messaging pattern can be applied by establishing a subscriber-publisher relationship between Service A and Service

B. This way, every time the data value is updated, an event is triggered and Service B, acting as the publisher, can notify Service A, which acts as the subscriber. The Asynchronous Queuing pattern can be applied between Service A and Service B so that the event notification message sent out by Service B will be received by Service A, even when Service A is unavailable.

B. The Event-Driven Messaging pattern can be applied by establishing a subscriber-publisher relationship between Service Consumer A and Service
A. This way, every time the data value is updated, an event is triggered and Service A, acting as the publisher, can notify Service Consumer A, which acts as the subscriber. The Asynchronous Queuing pattern can be applied between Service Consumer A and Service A so that the event notification message sent out by Service A will be received by Service Consumer A, even when Service Consumer A is unavailable.
C. The Asynchronous Queuing pattern can be applied so that messaging queues are established between Service A and Service B and between Service Consumer A and Service A. This way, messages are never lost due to the unavailability of Service A or Service B.
D. The Event-Driven Messaging pattern can be applied by establishing a subscriber-publisher relationship between Service Consumer A and a database monitoring agent introduced through the application of the Service Agent pattern. The database monitoring agent monitors updates made by the legacy client to the database. This way, every time the data value is updated, an event is triggered and the database monitoring agent, acting as the publisher, can notify Service Consumer A, which acts as the subscriber. The Asynchronous Queuing pattern can be applied between Service Consumer A and the database monitoring agent so that the event notification message sent out by the database monitoring agent will be received by Service Consumer A, even when Service Consumer A is unavailable.
Answer: A
Explanation:
This solution is the most appropriate one among the options presented. By using the Event-Driven Messaging pattern, Service A can be notified of changes to the data value without having to be invoked repeatedly by Service Consumer A, which reduces the resources required for message exchange. Asynchronous Queuing ensures that the event notification message is not lost due to the unavailability of Service A or Service B. This approach improves the efficiency of the service composition architecture.

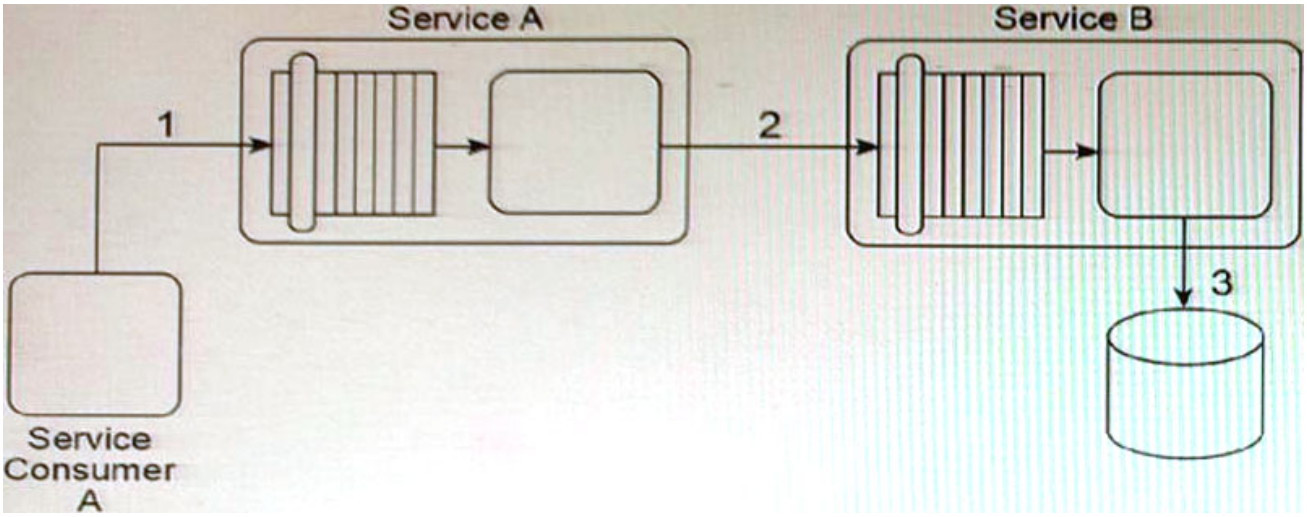
2.When Service A receives a message from Service Consumer A (1), the message is processed by Component A. This component first invokes Component B (2), which uses values from the message to query Database A in order to retrieve additional data. Component B then returns the additional data to Component A. Component A then invokes Component C (3), which interacts with the API of a legacy system to retrieve a new data value. Component C then returns the data value back to Component A.



Next, Component A sends some of the data It has accumulated to Component D (4), which writes the data to a text file that is placed in a specific folder. Component D then waits until this file is imported into a different system via a regularly scheduled batch import. Upon completion of the import, Component D returns a success or failure code back to Component A. Component A finally sends a response to Service Consumer A (5) containing all of the data collected so far and Service Consumer A writes all of the data to Database B (6). Components A, B, C, and D belong to the Service A service architecture. Database A, the legacy system and the file folders are shared resources within the IT enterprise. Service A is an entity service with a service architecture that has grown over the past few years. As a result of a service inventory-wide redesign project, you are asked to revisit the Service A service architecture in order to separate the logic provided by Components B, C, and D into three different utility services without disrupting the behavior of Service A as it relates to Service Consumer A.
What steps can be taken to fulfill these requirements?
A. The Legacy Wrapper pattern can be applied so that Component B is separated into a separate wrapper utility service that wraps the shared database. The Asynchronous Queuing pattern can be applied so that a messaging queue is positioned between Component A and Component C, thereby enabling communication during the times when the legacy system may be unavailable or heavily accessed by other parts of the IT enterprise. The Service Fagade pattern can be applied so that a fagade component is added between Component A and Component D so that any change In behavior can be compensated. The Service Autonomy principle can be further applied to Service A to help make up for any performance loss that may result from splitting the component into a separate wrapper utility service.

B. The Legacy Wrapper pattern can be applied so that Component B Is separated into a separate utility service that wraps the shared database. The Legacy Wrapper pattern can be applied again so that Component C is separated into a separate utility service that acts as a wrapper for the legacy system API. The Legacy Wrapper pattern can be applied once more to Component D so that it is separated into another utility service that provides standardized access to the file folder. The Service Fagade pattern can be applied so that three fagade components are added: one between Component A and each of the new wrapper utility services. This way, the fagade components can compensate for any change in behavior that may occur as a result of the separation. The Service Composability principle can be further applied to Service A and the three new wrapper utility services so that all four services are optimized for participation in the new service composition. This will help make up for any performance loss that may result from splitting the three components into separate services.
C. The Legacy Wrapper pattern can be applied so that Component B is separated into a separate utility service that wraps the shared database. The Legacy Wrapper pattern can be applied again so that Component C is separated into a separate utility service that acts as a wrapper for the legacy system API. Component D can also be separated into a separate service and the Event-Driven Messaging pattern can be applied to establish a publisher-subscriber relationship between this new service and Component A. The interaction between Service Consumer A and Component A can then be redesigned so that Component A first interacts with Component B and the new wrapper service. Service A then issues a final message back to Service Consumer A. The Service Composability principle can be further applied to Service A and the three new wrapper utility services so that all four services are optimized for participation in the new service composition. This will help make up for any performance loss that may result from splitting the three components into separate services.
D. The Legacy Wrapper pattern can be applied so that Component B is separated into a separate wrapper utility service that wraps the shared database. The State Repository and State Messaging patterns can be applied so that a messaging repository is positioned between Component A and Component C, thereby enabling meta data-driven communication during the times when the legacy system may be unavailable or heavily accessed by other parts of the IT enterprise. The Service Fagade pattern can be applied so that a fagade component is added between Component A and Component D so that any change in behavior can be compensated. The Service Statelessness principle can be further applied to Service A to help make up for any performance loss that may result from splitting the component into a separate wrapper utility service.
Answer: B

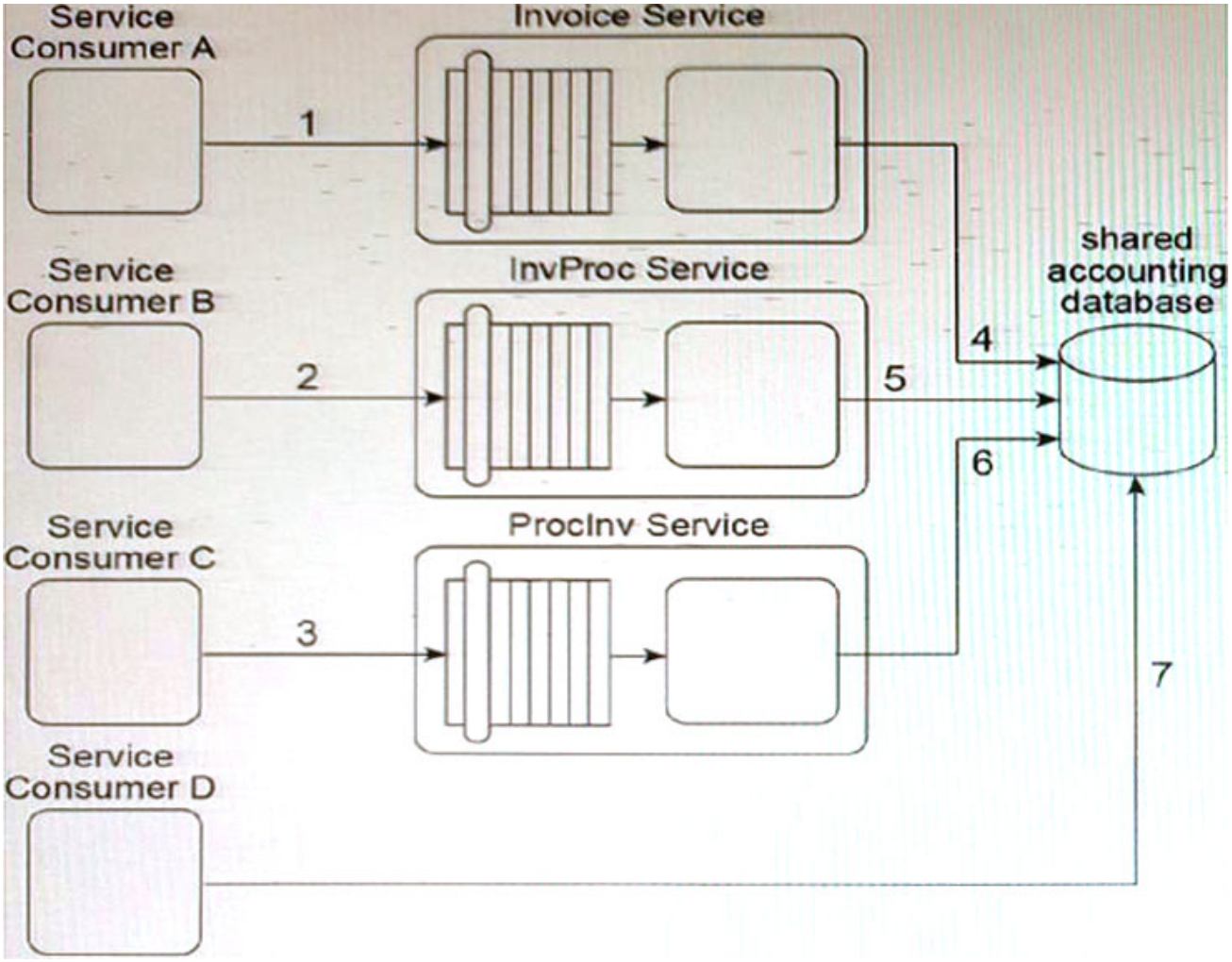
3.Service A is a SOAP-based Web service with a functional context dedicated to invoice-related processing. Service B is a REST-based utility service that provides generic data access to a database. In this service composition architecture, Service Consumer A sends a SOAP message containing an invoice XML document to Service A (1). Service A then sends the invoice XML document to Service B (2), which then writes the invoice document to a database (3).



The data model used by Service Consumer A to represent the invoice document is based on XML Schema A. The service contract of Service A is designed to accept invoice documents based on XML Schema
B. The service contract for Service B is designed to accept invoice documents based on XML Schema A. The database to which Service B needs to write the invoice record only accepts entire business documents in a proprietary Comma Separated Value (CSV) format.
Due to the incompatibility of the XML schemas used by the services, the sending of the invoice document from Service Consumer A through to Service B cannot be accomplished using the services as they currently exist. Assuming that the Contract Centralization pattern is being applied and that the Logic Centralization pattern is not being applied, what steps can be taken to enable the sending of the invoice document from Service Consumer A to the database without adding logic that will increase the runtime performance requirements?
A. Service Consumer A can be redesigned to use XML Schema B so that the SOAP message it sends is compliant with the service contract of Service A. The Data Model Transformation pattern can then be applied to transform the SOAP message sent by Service A so that it conforms to the XML Schema A used by Service B. The Standardized Service Contract principle must then be applied to Service B and Service Consumer A so that the invoice XML document is optimized to avoid unnecessary validation.
B. The service composition can be redesigned so that Service Consumer A sends the invoice document directly to Service B after the specialized invoice processing logic from Service A is copied to Service B. Because Service Consumer A and Service B use XML Schema A, the need for transformation logic is avoided. This naturally applies the Service Loose Coupling principle because Service Consumer A is not required to send the invoice document In a format that is compliant with the database used by Service B.
C. Service Consumer A can be redesigned to write the invoice document directly to the database. This reduces performance requirements by avoiding the involvement of Service A and Service B. It further supports the application of the Service Loose Coupling principle by ensuring that Service Consumer A contains data access logic that couples it directly to the database.
D. The service composition can be redesigned so that Service Consumer A sends the invoice document directly to Service B. Because Service Consumer A and Service B use XML Schema A, the need for transformation logic is avoided. This naturally applies the Logic Centralization pattern because Service Consumer A is not required to send the invoice document In a format that is compliant with the database used by Service B.

Answer: A
Explanation:
The recommended solution is to use the Data Model Transformation pattern to transform the invoice XML document from Schema B to Schema A before passing it to Service B. This solution maintains the separation of concerns and allows each service to work with its own specific XML schema. Additionally, the Standardized Service Contract principle should be applied to Service B and Service Consumer A to ensure that unnecessary validation is avoided, thus optimizing the invoice XML document. This solution avoids adding logic that will increase the runtime performance requirements.

4.Our service inventory contains the following three services that provide Invoice-related data access capabilities: Invoice, InvProc and Proclnv. These services were created at different times by different project teams and were not required to comply with any design standards. Therefore, each of these services has a different data model for representing invoice data.



Currently, each of these three services has a different service consumer: Service Consumer A accesses the Invoice service (1), Service Consumer B (2) accesses the InvProc service, and Service Consumer C (3) accesses the Proclnv service. Each service consumer invokes a data access capability of an invoice-related service, requiring that service to interact with the shared accounting database that is used by all invoice-related services (4, 5, 6).
Additionally, Service Consumer D was designed to access invoice data from the shared accounting

database directly (7). (Within the context of this architecture, Service Consumer D is labeled as a service consumer because it is accessing a resource that is related to the illustrated service architectures.) Assuming that the Invoice service, InvProc service and Proclnv service are part of the same service inventory, what steps would be required to fully apply the Official Endpoint pattern?

A. One of the invoice-related services needs to be chosen as the official service providing invoice data access capabilities. Service Consumers A, B, and C then need to be redesigned to only access the chosen invoice-related service. Because Service Consumer D does not rely on an invoice-related service, it is not affected by the Official Endpoint pattern and can continue to access the accounting database directly. The Service Abstraction principle can be further applied to hide the existence of the shared accounting database and other implementation details from current and future service consumers.

B. One of the invoice-related services needs to be chosen as the official service providing invoice data access capabilities and logic from the other two services needs to be moved to execute within the context of the official Invoice service. Service Consumers A, B, and C then need to be redesigned to only access the chosen invoice-related service. Service Consumer D also needs to be redesigned to not access the shared accounting database directly, but to also perform its data access by interacting with the official invoice-related service. The Service Abstraction principle can be further applied to hide the existence of the shared accounting database and other implementation details from current and future service consumers.

C. Because Service Consumers A, B, and C are already carrying out their data access via published contracts, they are not affected by the Official Endpoint pattern. Service Consumer D needs to be redesigned so that it does not access the shared accounting database directly, but instead performs its data access by interacting with the official invoice-related service. The Service Abstraction principle can be further applied to hide the existence of the shared accounting database and other implementation details from current and future service consumers.

D. One of the invoice-related services needs to be chosen as the official service providing invoice data access capabilities. Because Service Consumer D does not rely on an invoice-related service, it is not affected by the Official Endpoint pattern and can continue to access the accounting database directly. The Service Loose Coupling principle can be further applied to decouple Service Consumers A, B, and C from the shared accounting database and other implementation details.

Answer: B

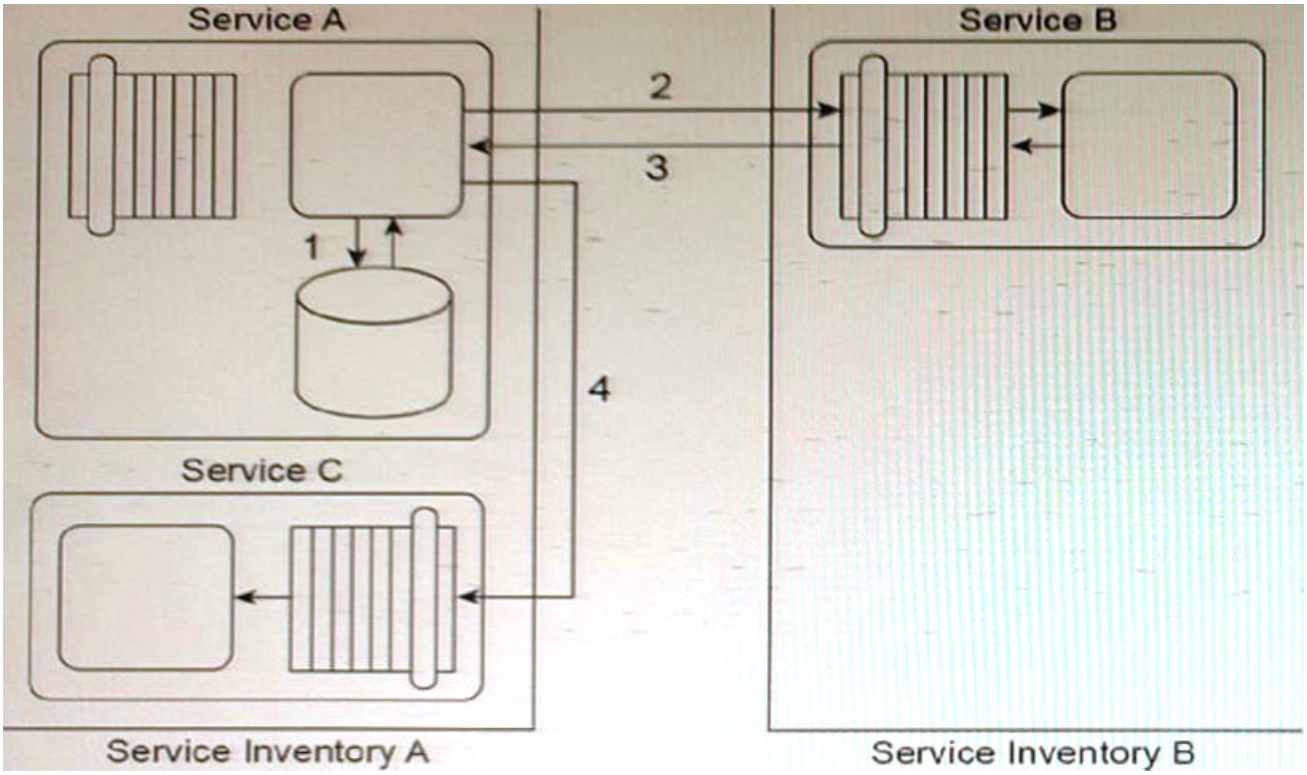
Explanation:

he Legacy Wrapper pattern can be applied so that Component B is separated into a separate utility service that wraps the shared database. The Legacy Wrapper pattern can be applied again so that Component C is separated into a separate utility service that acts as a wrapper for the legacy system API. The Legacy Wrapper pattern can be applied once more to Component D so that it is separated into another utility service that provides standardized access to the file folder. The Service Facade pattern can be applied so that three facade components are added: one between Component A and each of the new wrapper utility services. This way, the facade components can compensate for any change in behavior that may occur as a result of the separation. The Service Composability principle can be further applied to Service A and the three new wrapper utility services so that all four services are optimized for participation in the new service composition. This will help make up for any performance loss that may result from splitting the three components into separate services.

By applying the Legacy Wrapper pattern to separate Components B, C, and D into three different utility

services, the shared resources within the IT enterprise (Database A, the legacy system, and the file folders) can be properly encapsulated and managed by dedicated services. The Service Facade pattern can then be used to create a façade component between Component A and each of the new wrapper utility services, allowing them to interact seamlessly without affecting Service Consumer A's behavior. Finally, the Service Composability principle can be applied to ensure that Service A and the three new wrapper utility services are optimized for participation in the new service composition. This will help to mitigate any performance loss that may result from splitting the three components into separate services.

5.Service A is a task service that sends Service B a message (2) requesting that Service B return data back to Service A in a response message (3). Depending on the response received, Service A may be required to send a message to Service C (4) for which it requires no response.



Before it contacts Service B, Service A must first retrieve a list of code values from its own database (1) and then place this data into its own memory. If it turns out that it must send a message to Service C, then Service A must combine the data it receives from Service B with the data from the code value list in order to create the message it sends to Service C. If Service A is not required to invoke Service C, it can complete its task by discarding the code values.
Service A and Service C reside in Service Inventory A. Service B resides in Service Inventory B.
You are told that the services in Service Inventory A were designed with service contracts that are based on different design standards and technologies than the services In Service Inventory B. As a result, Service A is a SOAP-based Web service and Service B Is a REST service that exchanges JSON-formatted messages. Therefore, Service A and Service B cannot currently communicate. Furthermore, Service C is an agnostic service that is heavily accessed by many concurrent service consumers. Service C frequently reaches its usage thresholds, during which it is not available and messages sent to it are not received.
What steps can be taken to solve these problems?

A. The Data Model Transformation pattern can be applied by establishing an intermediate processing layer between Service A and Service B that can transform a message from one data model to another at runtime. The Intermediate Routing and Service Agent patterns can be applied so that when Service B sends a response message, a service agent can intercept the message and, based on its contents, either forward the message to Service A or route the message to Service C. The Service Autonomy principle can be further applied to Service C together with the Redundant Implementation pattern to help establish a more reliable and scalable service architecture.
B. The Data Format Transformation pattern can be applied by establishing an intermediate processing layer between Service A and Service B that can transform a message from one data format to another at runtime. The Asynchronous Queuing pattern can be applied to establish an intermediate queue between Service A and Service C so that when Service A needs to send a message to Service C, the queue will store the message and retransmit it to Service C until it is successfully delivered. The Service Autonomy principle can be further applied to Service C together with the Redundant Implementation pattern to help establish a more reliable and scalable service architecture.
C. The Data Model Transformation pattern can be applied by establishing an intermediate processing layer between Service A and Service B that can transform a message from one data model to another at runtime. The Intermediate Routing and Service Agent patterns can be applied so that when Service B sends a response message, a service agent can intercept the message and, based on its contents, either forward the message to Service A or route the message to Service C. The Service Statelessness principle can be applied with the help of the State Repository pattern so that Service A can write the code value data to a state database while it is waiting for Service B to respond.
D. The Data Format Transformation pattern can be applied by establishing an intermediate processing layer between Service A and Service B that can transform a message from one data format to another at runtime. The Asynchronous Queuing pattern can be applied to establish an intermediate queue between Service A and Service B so that when Service A needs to send a message to Service B, the queue will store the message and retransmit it to Service B until it is successfully delivered. The Service Reusability principle can be further applied to Service C together with the Redundant Implementation pattern to help establish a more reusable and scalable service architecture.

Answer: B

Explanation:

The problem is that Service A and Service B are using different technologies and cannot communicate. Therefore, an intermediate processing layer can be established that can transform messages from one data format to another at runtime. This can be achieved using the Data Format Transformation pattern.
Additionally, Service C frequently reaches its usage thresholds and is not always available, so an Asynchronous Queuing pattern can be applied to establish an intermediate queue between Service A and Service C. This queue will store the messages sent by Service A to Service C and retransmit them until they are successfully delivered. This approach improves the reliability of the system.
Moreover, the Redundant Implementation pattern can be applied to Service C to ensure its availability and scalability, and the Service Autonomy principle can be applied to make Service C independent of other services.