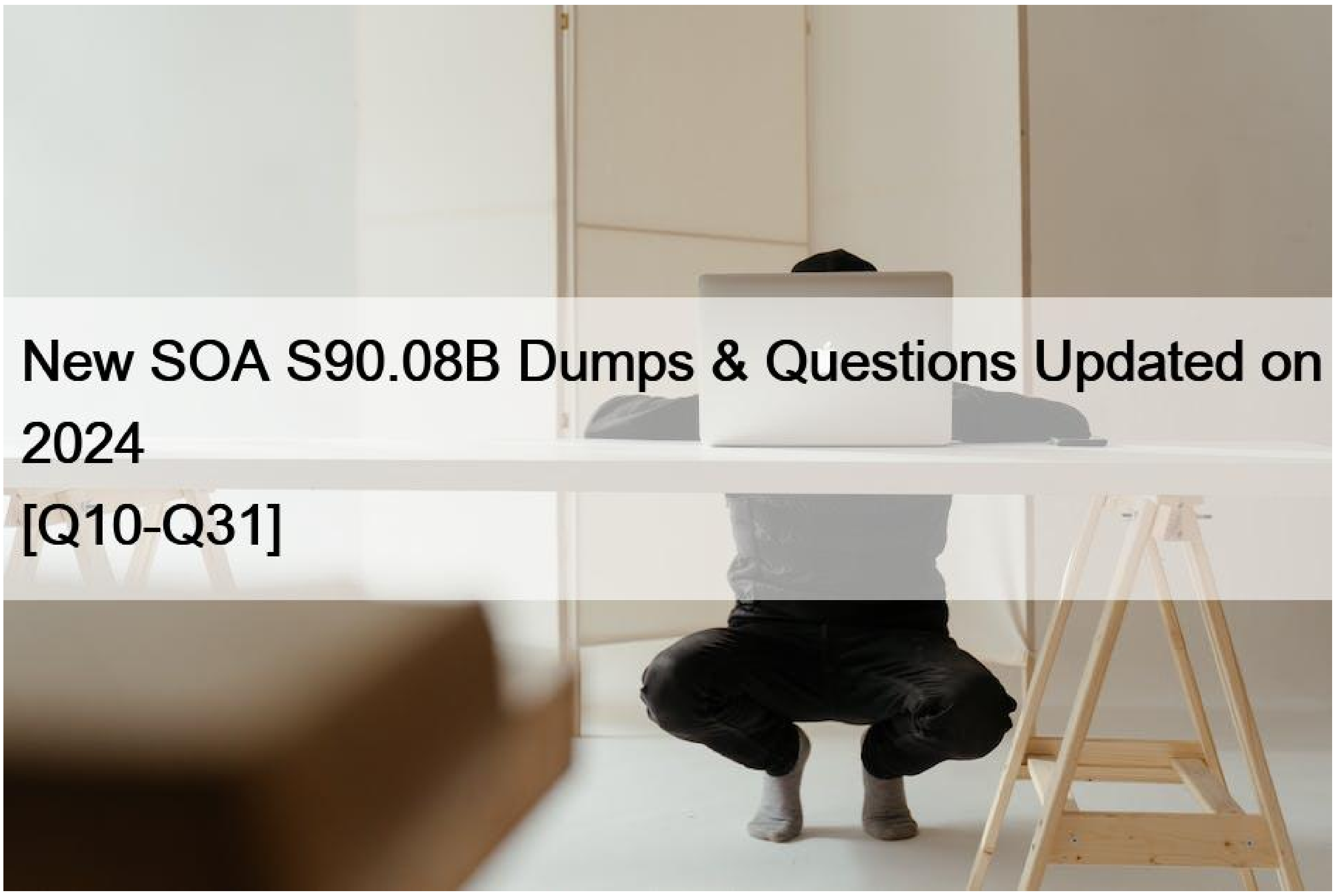
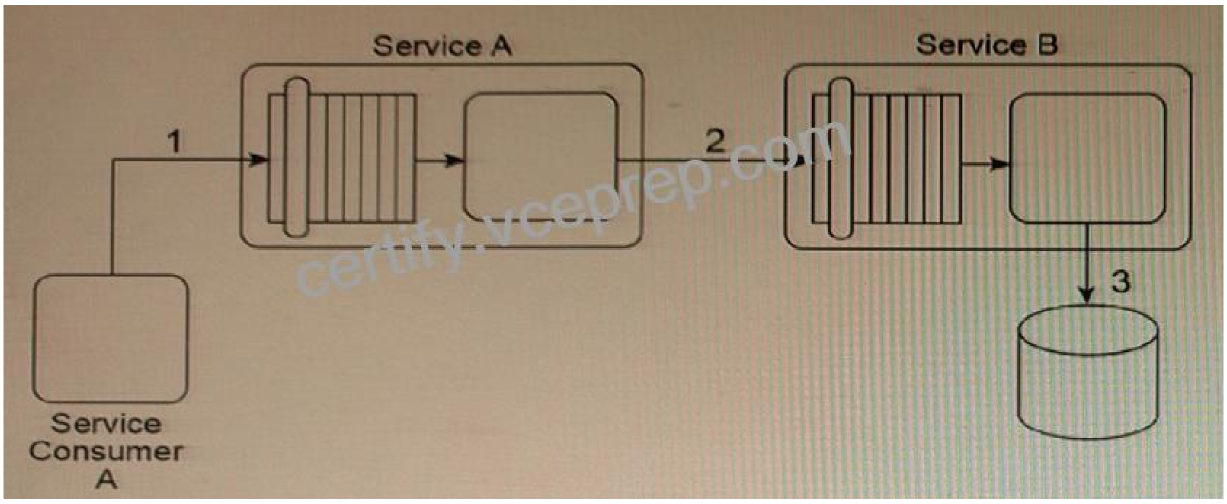
New SOA S90.08B Dumps & Questions Updated on 2024 [Q10-Q31



New SOA S90.08B Dumps & Questions Updated on 2024
Dumps to Pass your S90.08B Exam with 100% Real Questions and Answers

The S90.08B exam is a lab-based exam that is designed to test the practical skills of applicants. S90.08B exam consists of a series of hands-on exercises that require applicants to design and implement various SOA-based solutions using microservices. S90.08B exam is designed to challenge applicants and test their ability to work in a fast-paced, dynamic environment. Successful completion of the exam demonstrates that an applicant has the necessary skills and knowledge to design and implement complex SOA solutions using microservices, which is a highly sought-after skill in today's IT job market.

QUESTION 10



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.

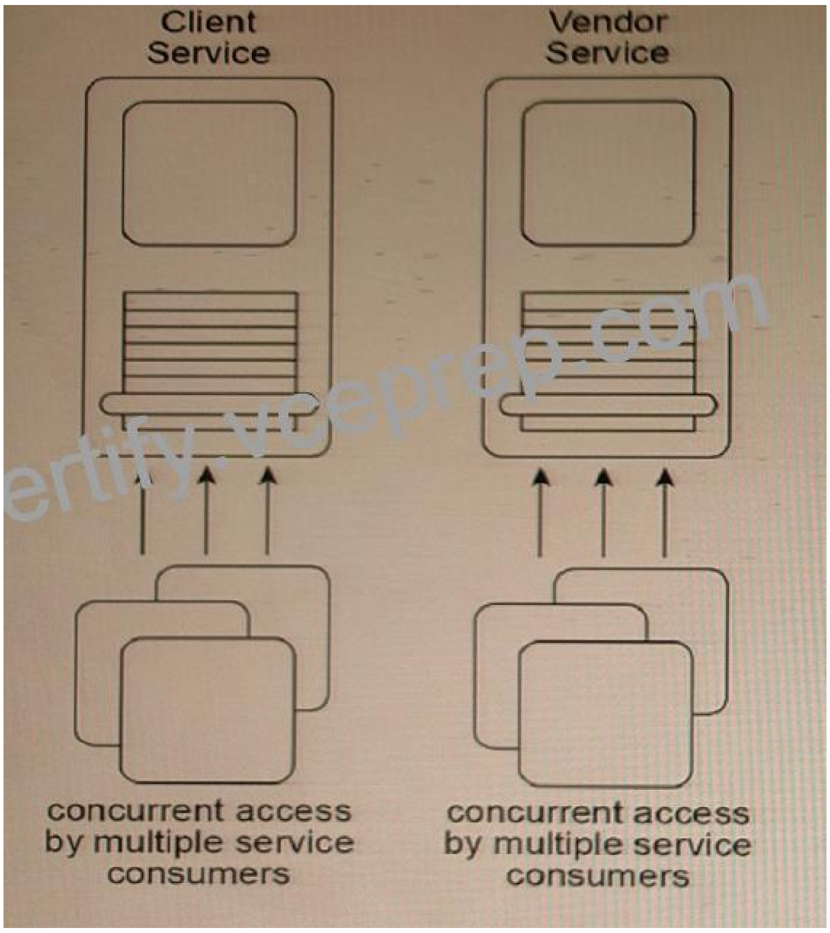
The service contract of Service A is designed to accept invoice documents based on XML Schema B. The service contract for Service B is designed to accept invoice documents based on XML Schema A. The database to which Service B needs to write the invoice record only accepts entire business documents in 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?
* 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.
* 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
* 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.
* 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.
* 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.
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 toensure that unnecessary validation is avoided, thus optimizing the invoice XML document. This solution avoids adding logic that will increase the runtime performance requirements.

QUESTION 11


Client Service
Vendor Service
concurrent access by multiple service consumers
concurrent access by multiple service consumers

The Client and Vendor services are agnostic services that are both currently part of multiple service compositions. As a result, these services are sometimes subjected to concurrent access by multiple service consumers.

The Client service primarily provides data access logic to a client database but also coordinates with other services to determine a clients credit rating. The Vendor service provides some data access logic but can also generate various dynamic reports based on specialized business requirements.

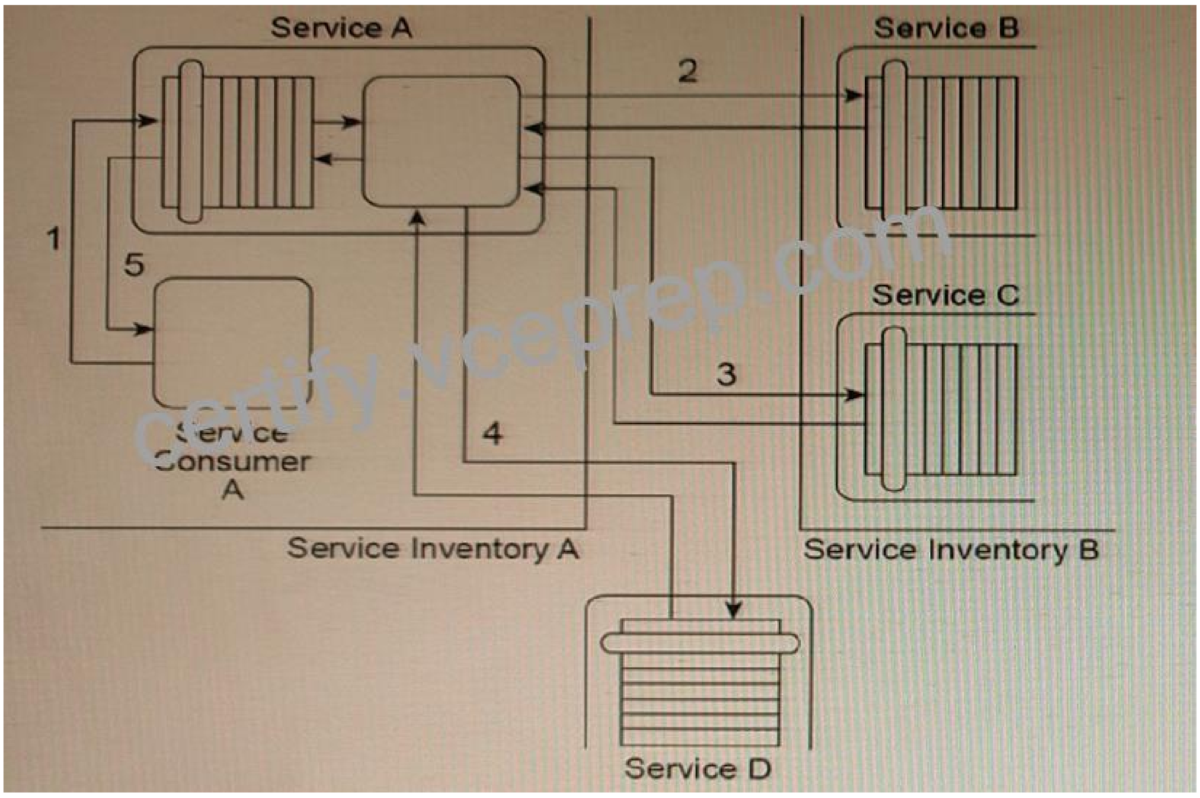
After reviewing historical statistics about the runtime activity of the two services, it is discovered that the Client service is serving an ever-increasing number of service consumers. It is regularly timing out, which in turn increases its call rate as service consumers retry their requests. The Vendor serviceoccasionally has difficulty meeting its service-level agreement (SLA) and when this occurs, penalties are assessed.

Recently, the custodian of the Client service was notified that the Client service will be made available to new service consumers external to its service inventory. The Client service will be providing free credit rating scores to any service consumer that connects to the service via the Internet. The Vendor service will remain internal to the service inventory and will not be exposed to external access.

Which of the following statements describes a solution that addresses these issues and requirements?
* The API Gateway pattern, together with the Inventory Endpoint pattern, can be applied to the service inventory to establish an inventory endpoint service and an intermediary layer of processing that will be accessed by external service consumers and that will interact with the Client service to process external service consumer requests. The Redundant Implementation pattern can be applied to both the Client and Vendor services to increase their availability and scalability.
* The Official Endpoint pattern can be applied to the Client service to establish a managed endpoint for consumption by service consumers external to the service inventory. The Concurrent Contracts pattern can be applied to the Vendor service, enabling it to connect with alternative Client service implementation, should the first attempt to connect fail.
* The State Repository pattern can be applied to the Client and Vendor services to establish a central statement management database that can be used to overcome runtime performance problems. The Official Endpoint pattern can be further applied to increase the availability and scalability of the Client service for service consumers external to the service inventory.
* The Microservice Deployment pattern is applied to the Client service to improve its autonomy and responsiveness to a greater range of service consumers. The Containerization pattern is applied to the Vendor service to establish a managed environment with a high degree of isolation for its report-related processing. The Endpoint Redirection pattern is further applied to ensure that request messages from service consumers outside of the service inventory are redirected away from the Client service.
Explanation

This solution addresses the specific requirements and issues identified in the scenario. The Official Endpoint pattern can be applied to the Client service to establish a managed endpoint for consumption by service consumers external to the service inventory, which will allow for controlled and managed access to the service. The Concurrent Contracts pattern can be applied to the Vendor service, which will enable it to connect with alternative Client service implementation if the first attempt to connect fails, thereby increasing its availability and reducing the possibility of penalties being assessed due to not meeting its SLA.

QUESTION 12



Service Consumer A and Service A reside in Service Inventory A. Service B and Service C reside in Service Inventory B. Service D is a public service that can be openly accessed via the World Wide Web. The service is also available for purchase so that it can be deployed independently within IT enterprises. Due to the rigorous application of the Service Abstraction principle within Service Inventory B, the only information that is made available about Service B and Service C are the published service contracts. For Service D, the service contract plus a service level agreement (SLA) are made available. The SLA indicates that Service D has a planned outage every night from 11:00pm to midnight.

You are an architect with a project team that is building services for Service Inventory A. You are told that the owners of Service Inventory A and Service Inventory B are not generally cooperative or communicative.

Cross-inventory service composition is tolerated, but not directly supported. As a result, no SLAs for Service B and Service C are available and you have no knowledge about how available these services are. Based on the service contracts you can determine that the services in Service Inventory B use different data models and a different transport protocol than the services in Service Inventory A. Furthermore, recent testing results have shown that the performance of Service D is highly unpredictable due to the heavy amount of concurrent access it receives from service consumers from other organizations. You are also told that there is a concern over how long Service Consumer A will need to remain stateful while waiting for a response from Service A.

What steps can be taken to solve these problems?
* The Event-Driven Messaging pattern can be applied to establish a subscriber-publisher relationship between Service Consumer A and Service A. This gives Service A the flexibility to provide its response to Service Consumer A whenever it is able to collect the three data values without having to require that Service Consumer A remain stateful. The Asynchronous Queuing pattern can be applied to position a central messaging queue between Service A and Service B and between Service A and Service C. The Data Model Transformation and Protocol Bridging patterns can be applied to enable communication between Service A and Service B and between Service A and Service C. The Redundant Implementation pattern can be applied so that a copy of Service D is brought in-house and made part of Service Inventory A.
* The Asynchronous Queuing pattern can be applied to position a central messaging queue between Service A and Service B and

between Service A and Service C and so that a separate messaging queue is positioned between Service A and Service Consumer A. The Data Model Transformation and Protocol Bridging patterns can be applied to enable communication between Service A and Service B and between Service A and Service C. The Redundant Implementation pattern can be applied so that a copy of Service D is brought in-house. The Legacy Wrapper pattern can be further applied to wrap Service D with a standardized service contract that is in compliance with the design standards used in Service Inventory A.
* The Containerization pattern can be applied to establish an environment for Service A to perform its processing autonomously. This gives Service A the flexibility to provide Service Consumer A with response messages consistently. The Asynchronous Queuing pattern can be applied so that a central messaging queue is positioned between Service A and Service B, between Service A and Service C, and between Service A and Service D. The Data Model Transformation and Protocol Bridging patterns can be applied to enable communication between Service A and Service B and between Service A and Service C.
* The Asynchronous Queuing pattern can be applied to position a message queue between Service A and Service B, between Service A and Service C, and between Service A and Service D. Additionally, a separate messaging queue is positioned between Service A and Service Consumer A. The Data Model Transformation and Protocol Bridging patterns can be applied to enable communication between Service A and Service B, between Service A and Service C, and between Service A and Service D. The Redundant Implementation pattern can be applied so that a copy of Service D is brought in-house. The Legacy Wrapper pattern can be further applied to wrap Service D with a standardized service contract that is in compliance with the design standards used in Service Inventory B.
Explanation

The Asynchronous Queuing pattern is applied to position a messaging queue between Service A, Service B, Service C, Service D, and Service Consumer A. This ensures that messages can be passed between these services without having to be in a stateful mode.

The Data Model Transformation and Protocol Bridging patterns are applied to enable communication between Service A and Service B, Service A and Service C, and Service A and Service D, despite their different data models and transport protocols.

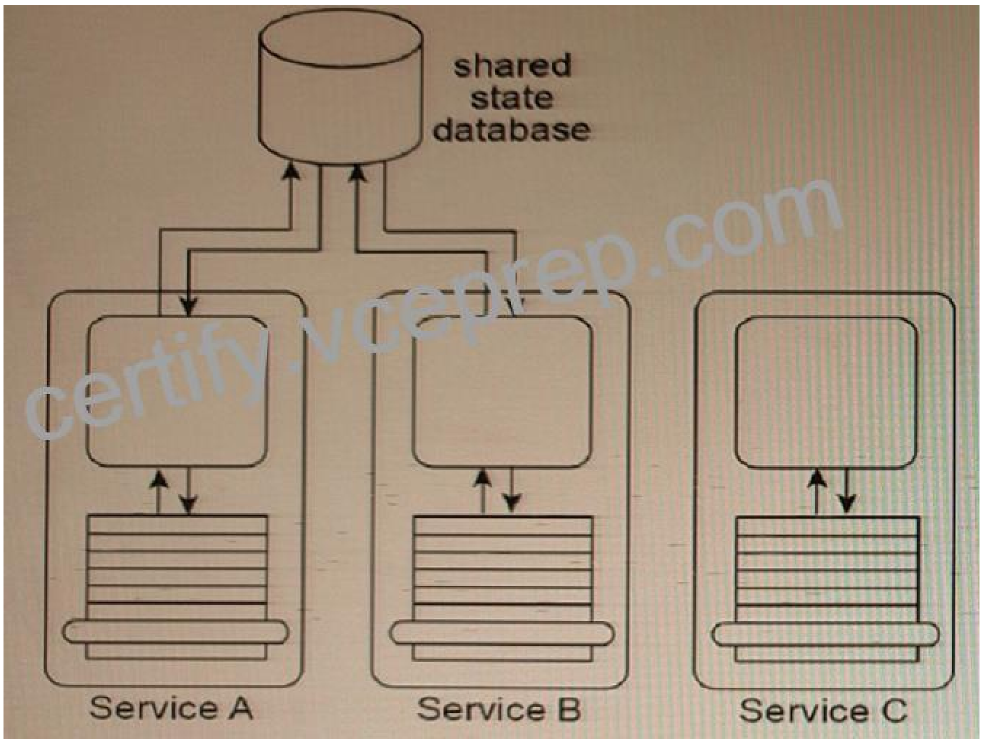
The Redundant Implementation pattern is applied to bring a copy of Service D in-house to ensure that it can be accessed locally and reduce the unpredictability of its performance.

The Legacy Wrapper pattern is applied to wrap Service D with a standardized service contract that complies with the design standards used in Service Inventory B. This is useful for service consumers who want to use Service D but do not want to change their existing applications or service contracts.

Overall, this approach provides a comprehensive solution that addresses the issues with Service A, Service B, Service C, and Service D, while maintaining compliance with the Service Abstraction principle.

QUESTION 13

Refer to Exhibit.



Services A, B, and C are non-agnostic task services. Service A and Service B use the same shared state database to defer their state data at runtime.

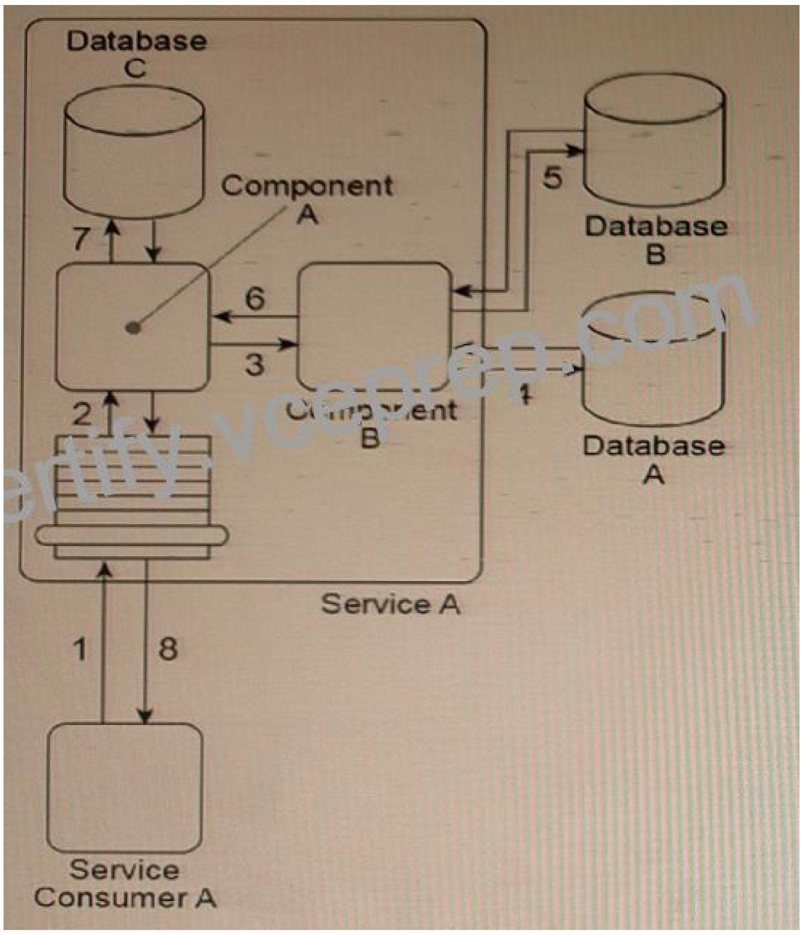
An assessment of the three services reveals that each contains some agnostic logic that cannot be made available for reuse because it is bundled together with non-agnostic logic.

The assessment also determines that because Service A, Service B and the shared state database are each located in physically separate environments, the remote communication required for Service A and Service B to interact with the shared state database is causing an unreasonable decrease in runtime performance.

How can the application of the Orchestration pattern improve this architecture?
* The application of the Orchestration pattern will result in an environment whereby the Official Endpoint, State Repository, and Service Data Replication patterns are automatically applied, allowing the shared state database to be replicated via official service endpoints for Services A and B so that each task service can have its own dedicated state database.
* The application of the Orchestration pattern will result in an environment whereby the non-agnostic logic can be cleanly separated from the agnostic logic that exists in Services A, B, and C, resulting in the need to design new agnostic services with reuse potential assured through the application of the Service Reusability principle. The State Repository pattern, which is supported by and local to the orchestration environment, provides a central state database that can be shared by Services A and B. The local state database avoids problems with remote communication.
* The application of the Orchestration pattern will result in an environment whereby the Compensating Service Transaction is automatically applied, resulting In the opportunity to create sophisticated exception logic that can be used to compensate for the performance problems caused by Services A and B having to remotely access the state database. The API Gateway and Service Broker patterns are also automatically applied, providing common transformation functions in a centralized processing layer to help overcome any disparity in the service contracts that will need to be created for the new agnostic services.
* The Orchestration pattern is not applicable to this architecture because it does not support the hosting of the required state repository.
The application of the Orchestration pattern can improve this architecture by cleanly separating the non-agnostic logic from the

agnostic logic, allowing the design of new agnostic services with reuse potential. The State Repository pattern, which is supported by and local to the orchestration environment, provides a central state database that can be shared by Services A and B. The local state database avoids problems with remote communication. Additionally, the Orchestration pattern provides a central controller that can coordinate the interactions between Services A, B, and C, reducing the need for remote communication between services and improving runtime performance.

QUESTION 14



Service Consumer A sends Service A a message containing a business document (1). The business document is received by Component A, which keeps the business document in memory and forwards a copy to Component B (3). Component B first writes portions of the business document to Database A (4). Component B then writes the entire business document to Database B and uses some of the data values from the business document as query parameters to retrieve new data from Database B (5).

Next, Component B returns the new date* back to Component A (6), which merges it together with the original business document it has been keeping in memory and then writes the combined data to Database C (7). The Service A service capability invoked by Service Consumer A requires a synchronousrequest-response data exchange. Therefore, based on the outcome of the last database update, Service A returns a message with a success or failure code back to Service Consumer A (8).

Databases A and B are shared, and Database C is dedicated to the Service A service architecture.

There are several problems with this architecture. The business document that Component A is required to keep in memory (while it waits for Component B to complete its processing) can be very large. The amount of runtime resources Service A uses to keep this data in memory can decrease the overall performance of all service instances, especially when it is concurrently invoked by multiple service consumers. Additionally, Service A can take a long time to respond back to Service Consumer A because Database A is a shared database that sometimes takes a long time to respond to Component B. Currently, Service Consumer A will wait for up to 30 seconds for a response, after which it will assume the request to Service A has failed and any subsequent response messages from Service A will be rejected.

What steps can be taken to solve these problems?
* The Service Statelessness principle can be applied together with the State Repository pattern to extend Database C so that it also becomes a state database allowing Component A to temporarily defer the business document data while it waits for a response from Component B. The Service Autonomy principle can be applied together with the Legacy Wrapper pattern to isolate Database A so that it is encapsulated by a separate wrapper utility service. The Compensating Service Transaction pattern can be applied so that whenever Service A&#8217;s response time exceeds 30 seconds, a notification is sent to a human administrator to raise awareness of the fact that the eventual response of Service A will be rejected by Service Consumer A.
* The Service Statelessness principle can be applied together with the State Repository pattern to establish a state database to which Component A can defer the business document data to while it waits for a response from Component B. The Service Autonomy principle can be applied together with the Service Data Replication pattern to establish a dedicated replicated database for Component B to access instead of shared Database A. The Asynchronous Queuing pattern can be applied to establish a message queue between Service Consumer A and Service A so that Service Consumer A does not need to remain stateful while it waits for a response from Service A.
* The Service Statelessness principle can be applied together with the State Repository pattern to establish a state database to which Component A can defer the business document data while it waits for a response from Component B. The Service Autonomy principle can be applied together with the Service Abstraction principle, the Legacy Wrapper pattern, and the Service Fagade pattern in order to isolate Database A so that it is encapsulated by a separate wrapper utility service and to hide the Database A implementation from Service A and to position a fagade component between Component B and the new wrapper service. This fagade component will be responsible for compensating the unpredictable behavior of Database A.
* None of the above.
Explanation

The problems with the current architecture can be addressed by applying the following patterns:

* Service Statelessness principle and State Repository pattern &#8211; This pattern allows Component A to defer the business document data to a state database while it waits for a response from Component B. This helps reduce the amount of runtime resources Service A uses to keep the data in memory and improves overall performance.

* Service Autonomy principle and Service Data Replication pattern &#8211; This pattern allows Component B to

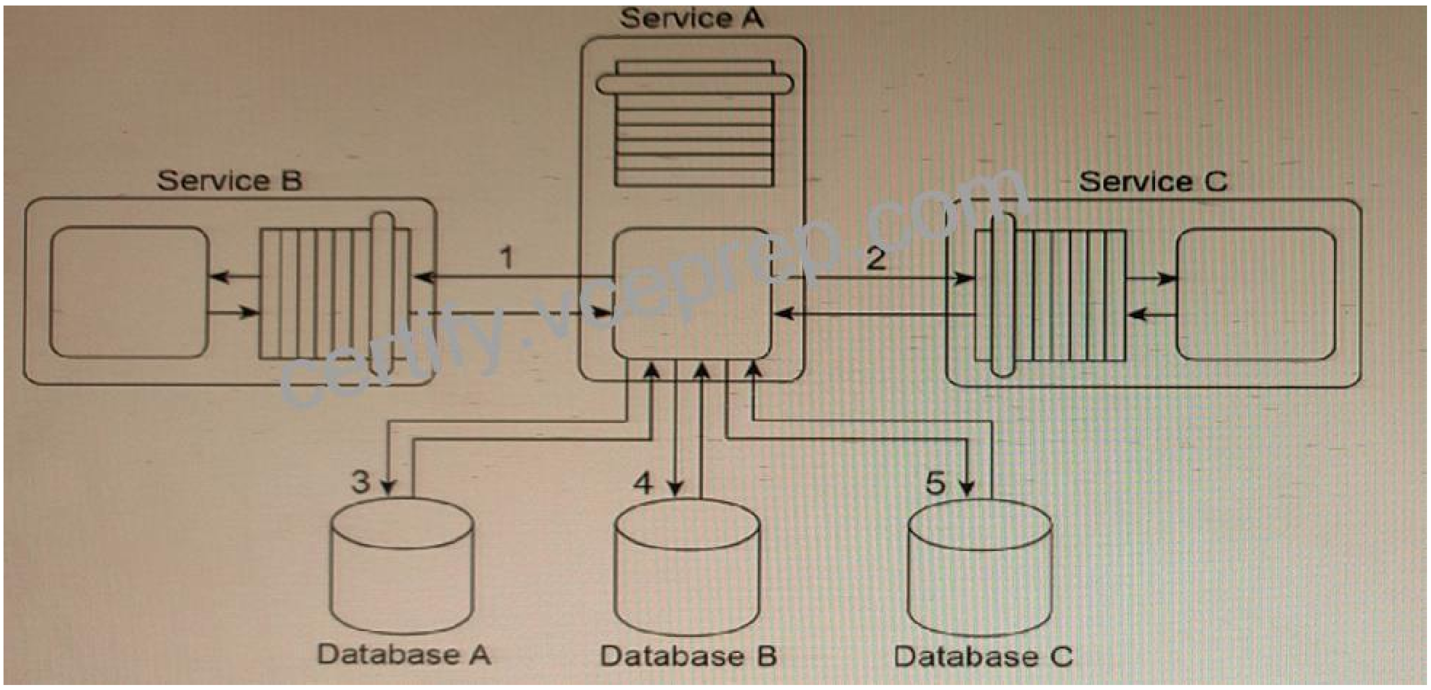
* access a dedicated replicated database instead of the shared Database A, which can improve response time.

* Asynchronous Queuing pattern &#8211; This pattern allows Service A to use a message queue to communicate with Service Consumer A asynchronously. This means that Service Consumer A does not need to remain stateful while waiting for a response from Service A, which can improve overall performance and scalability.

Therefore, option B is the correct answer. Option A is incorrect because it suggests using the Compensating Service Transaction pattern to raise awareness of the eventual response rejection, which does not actually solve the problem. Option C is also incorrect because it suggests using multiple patterns, which may not be necessary and can add unnecessary complexity to the architecture.

QUESTION 15

Refer to Exhibit.



Service A is an entity service that provides a set of generic and reusable service capabilities. In order to carry out the functionality of any one of its service capabilities, Service A is required to compose Service B (1) and Service C (2), and Service A is required to access Database A (3), Database B (4), and Database C (5). These three databases are shared by other applications within the IT enterprise.

All of service capabilities provided by Service A are synchronous, which means that for each request a service consumer makes, Service A is required to issue a response message after all of the processing has completed.

Service A is one of many entity services that reside In a highly normalized service Inventory. Because Service A provides agnostic logic, it is heavily reused and is currently part of many service compositions.

You are told that Service A has recently become unstable and unreliable. The problem has been traced to two issues with the current service architecture. First, Service B, which Is also an entity service, is being increasingly reused and has itself become unstable and unreliable. When Service B fails, the failure is carried over to Service A.

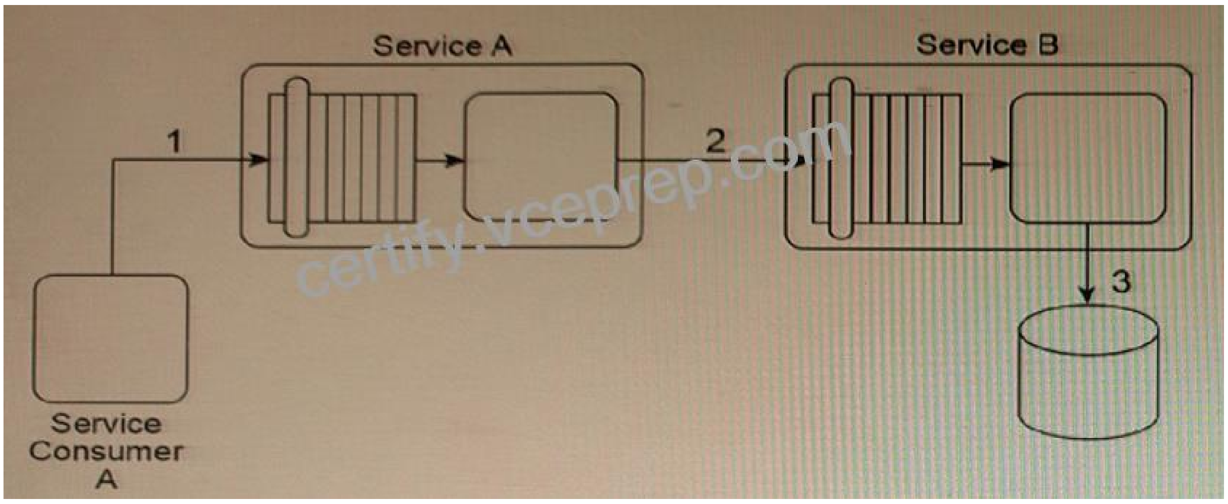
Secondly, shared Database B has a complex data model. Some of the queries issued by Service A to shared Database B can take a very long time to complete.

What steps can be taken to solve these problems without compromising the normalization of the service inventory?
* The Redundant Implementation pattern can be applied to Service A, thereby making duplicate deployments of the service available. This way, when one implementation of Service A is too busy, another implementation can be accessed by service consumers instead. The Service Data Replication pattern can be applied to establish a dedicated database that contains an exact copy of the data from shared Database B that is required by Service A.
* The Redundant Implementation pattern can be applied to Service B, thereby making duplicate deployments of the service available. This way, when one implementation of Service B is too busy, another implementation can be accessed by Service A instead. The Data Model Transformation pattern can be applied to establish a dedicated database that contains an exact copy of the data from shared Database B that is required by Service A.
* The Redundant Implementation pattern can be applied to Service B, thereby making duplicate deployments of the service available. This way, when one implementation of Service B is too busy, another implementation can be accessed by Service A

instead. The Service Data Replication pattern can be applied to establish a dedicated database that contains a copy of the data from shared Database B that is required by Service A. The replicated database is designed with an optimized data model to improve query execution performance.
* The Redundant Implementation pattern can be applied to Service A, thereby making duplicate deployments of the service available. This way, when one implementation of Service A is too busy, another implementation can be accessed by service consumers instead. The Service Statelessness principle can be applied with the help of the State Repository pattern In order to establish a state database that Service A can use to defer state data it may be required to hold for extended periods, thereby improving its availability and scalability.
This solution addresses both issues with the current service architecture. By applying the Redundant Implementation pattern to Service B, duplicate deployments of the service are made available, ensuring that when one implementation fails, another can be accessed by Service A. Additionally, the Service Data Replication pattern can be applied to establish a dedicated database that contains a copy of the data from shared Database B that is required by Service A. This replicated database is designed with an optimized data model to improve query execution performance, ensuring that queries issued by Service A to the database can complete more quickly, improving the overall stability and reliability of Service A. By applying these patterns, the problems with Service A can be solved without compromising the normalization of the service inventory.

QUESTION 16

Refer to Exhibit.



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?
* 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.
* 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.
* 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.

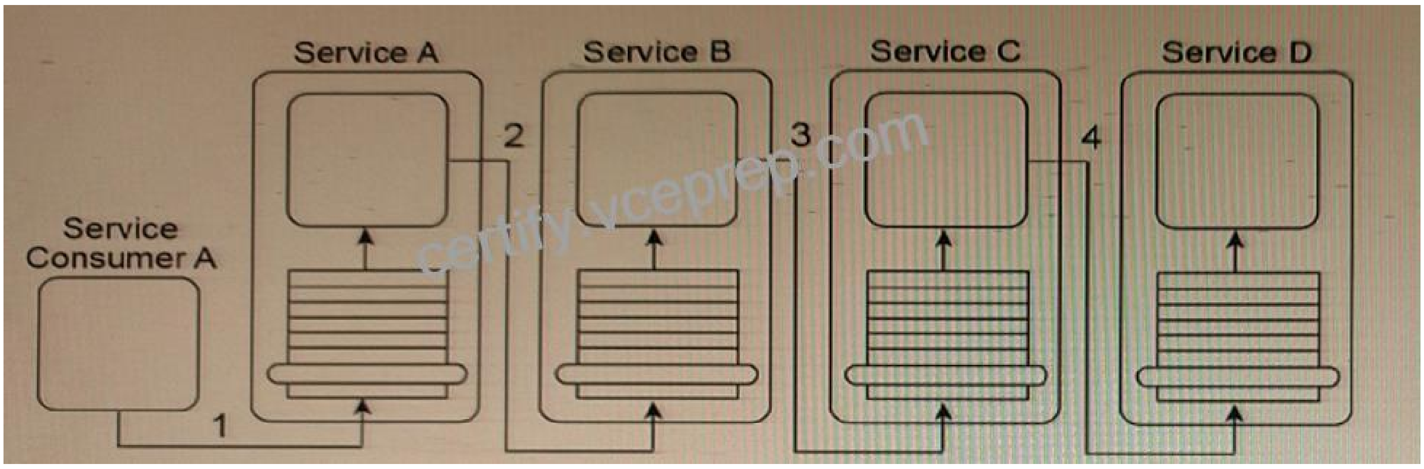
It further supports the application of the Service Loose Coupling principle by ensuring that Service Consumer A contains data access logic that couples it directly to the database.
* The service composition can be redesigned so that Service Consumer A sends the invoice document directly to Service B.

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.
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.

QUESTION 17

Refer to Exhibit.



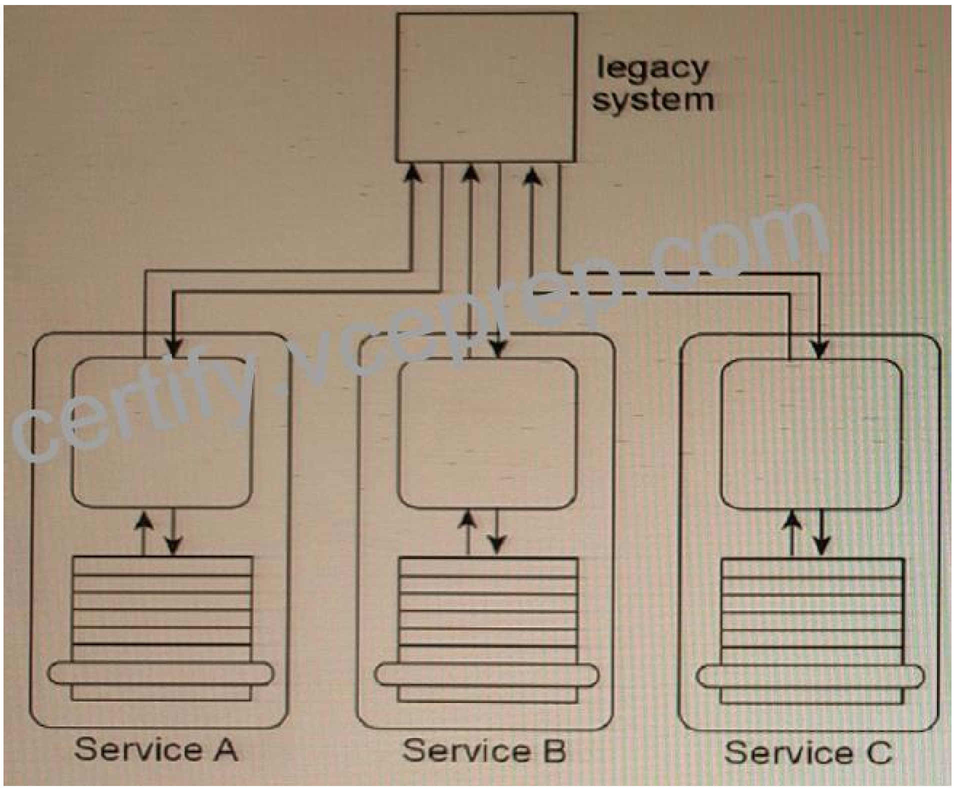
Service Consumer A sends a message to Service A (1), which then forwards the message to Service B (2). Service B forwards the message to Service C (3), which finally forwards the message to Service D (4). However, Services A, B and C each contain logic that reads the contents of the message to determine what intermediate processing to perform and which service to forward the message to. As a result, what is shown in the diagram is only one of several possible runtime scenarios.

Currently, this service composition architecture is performing adequately, despite the number of services that can be involved in the transmission of one message. However, you are told that new logic is being added to Service A that will require it to compose one other service to retrieve new data at runtime that Service A will need access to in order to determine where to forward the message to. The involvement of the additional service will make the service composition too large and slow.

What steps can be taken to improve the service composition architecture while still accommodating the new requirements and avoiding an increase in the amount of service composition members?
* The Service Instance Routing pattern can be applied to introduce a Routing service to provide a centralized service to contain routing-related business rules. This new Routing service can be accessed by Service A and Service C so they can determine where to forward messages to at runtime. The Service Reusability principle can be further applied to ensure that the logic in all remaining services is designed to be multi-purpose and reusable.
* The Asynchronous Queuing pattern can be applied together with a Routing service that is invoked by messages read from a messaging queue. This new Routing service can replace Service B and can be accessed by Service A and Service C so they can determine where to forward messages to at runtime. The Service Loose Coupling principle can be further applied to ensure that the new Routing service remains decoupled from other services so that it can perform its routing functions independently from service contract invocation.
* The Intermediate Routing pattern can be applied together with the Service Agent pattern by removing Service B or Service C from the service composition and replacing it with a service agent capable of intercepting and forwarding the message at runtime based on pre-defined routing logic. The Service Discoverability principle can be further applied to ensure that Service A can be found by any future service consumers.
* The Intermediate Routing pattern can be applied together with the Service Agent pattern to establish a service agent capable of intercepting and forwarding the message at runtime based on pre-defined routing logic. The Service Composability principle can be further applied to ensure that all services are designed as effective service composition participants.

This solution addresses the issue of the service composition becoming too large and slow by introducing a new Routing service that is invoked by messages read from a messaging queue. This allows Service A and Service C to determine where to forward messages to at runtime without the need for additional services in the composition. The Service Loose Coupling principle is applied to ensure that the new Routing service remains decoupled from other services so that it can perform its routing functions independently from service contract invocation.

QUESTION 18



Service A, Service B, and Service Care entity services, each designed to access the same shared legacy system.

Service A manages order entities, Service B manages invoice entities, and Service C manages customer entities. Service A, Service B, and Service C are REST services and are frequently reused by different service compositions. The legacy system uses a proprietary file format that Services A, B, and C need to convert to and from.

You are told that compositions involving Service A, Service B, and Service C are unnecessarily complicated due to the fact that order, invoice, and customer entitles are all related to each other. For example, an order has a customer, an invoice has an order, and so on. This results In calls to multiple services to reconstruct a complete order document. You are asked to architect a solution that will simplify the composition logic by minimizing the number of services required to support simple businessfunctions like order management or bill payment. Additionally, you are asked to reduce the amount of redundant data transformation logic that is found in Services A, B, and C.

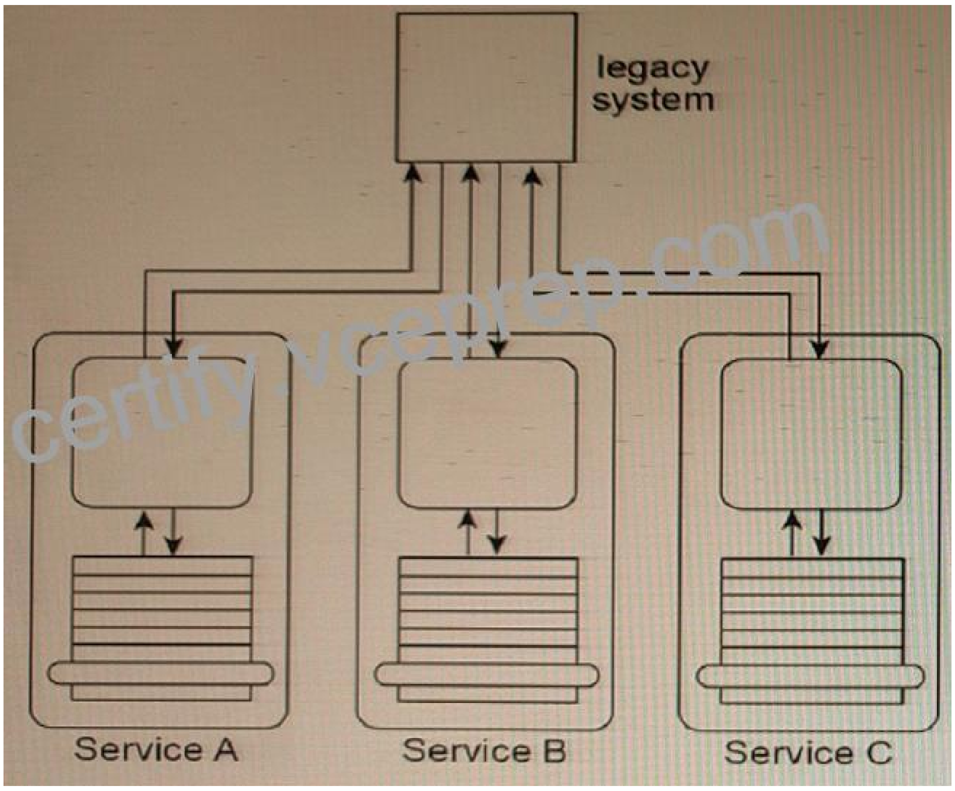
How will you accomplish these goals?
* The Enterprise Service Bus pattern can be applied to introduce an intermediate processing layer between Services A, B, and C and the legacy system. The enterprise service bus can be used to consolidate and execute the necessary transformation logic currently held within the services. The Endpoint Redirection pattern can be applied to re-route calls from one service to another to provide access to related entity data.
* The Legacy Wrapper pattern can be applied to create a service to expose the legacy system through a standardized service contract. The core logic of the wrapping service would provide all necessary data transformation functionality to convert between inventory-standardized data representations and the proprietary format. The Lightweight Endpoint pattern can be applied to establish lightweight capabilities that can return related entity data directly to service consumers.
* The Enterprise Service Bus pattern can be applied to introduce an intermediate processing layer between Services A, B, and C and the legacy system. The enterprise service bus can be used to consolidate and execute the transformation logic currently held within the services. The Content Negotiation pattern can be applied to return a content link to related entity data to a service consumer, which allows for simpler and more dynamic composition logic. The service consumer effectively invokes the relevant service

through the returned link to obtain the related entity data.
* The Legacy Wrapper pattern can be applied to create a service to expose the legacy system through a standardized service contract. The core logic of the wrapping service would provide all necessary data transformation functionality to convert between inventory-standardized data representations and the proprietary format. The Endpoint Redirection pattern can be applied to return a link to related entity data to a service consumer, which allows for simpler and more dynamic composition logic. The service consumer effectively invokes the relevant service through the returned link to obtain the related entity data.
Explanation

The Lightweight Endpoint pattern can be applied to establish lightweight capabilities that can return related entity data directly to service consumers, simplifying the composition logic by minimizing the number of services required to support simple business functions like order management or bill payment. This approach provides a standardized and simplified interface for the legacy system, reducing the complexity of the integration process with the entity services, and enabling them to focus on their core functionality.

QUESTION 19

Refer to Exhibit.

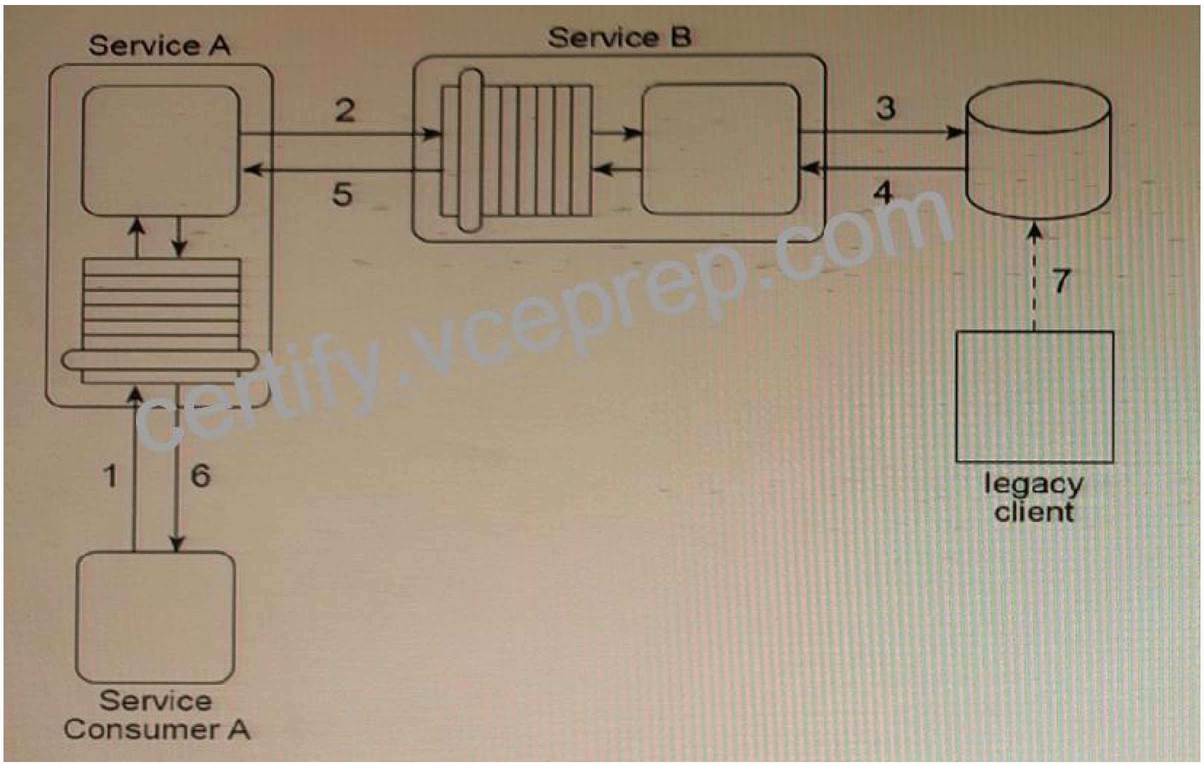


Service A, Service B, and Service C are entity services, each designed to access the same shared legacy system. Service A manages order entities, Service B manages invoice entities, and Service C manages customer entities. Service A, Service B, and Service C are REST services and are frequently reused by different service compositions. The legacy system uses a proprietary file format that Services A, B, and C need to convert to and from.

You are told that compositions involving Service A, Service B, and Service C are unnecessarily complicated due to the fact that order, invoice, and customer entitles are all related to each other. For example, an order has a customer, an invoice has an order, and so on. This results In calls to multiple services to reconstruct a complete order document. You are asked to architect a solution that will simplify the composition logic by minimizing the number of services required to support simple business functions like order management or bill payment. Additionally, you are asked to reduce the amount of redundant data transformation logic that is found in Services A, B, and C.

How will you accomplish these goals?
* The Enterprise Service Bus pattern can be applied to introduce an intermediate processing layer between Services A, B, and C and the legacy system. The enterprise service bus can be used to consolidate and execute the necessary transformation logic currently held within the services. The Endpoint Redirection pattern can be applied to re-route calls from one service to another to provide access to related entity data.
* The Legacy Wrapper pattern can be applied to create a service to expose the legacy system through a standardized service contract. The core logic of the wrapping service would provide all necessary data transformation functionality to convert between inventory-standardized data representations and the proprietary format. The Lightweight Endpoint pattern can be applied to establish lightweight capabilities that can return related entity data directly to service consumers.
* The Enterprise Service Bus pattern can be applied to introduce an intermediate processing layer between Services A, B, and C and the legacy system. The enterprise service bus can be used to consolidate and execute the transformation logic currently held within the services. The Content Negotiation pattern can be applied to return a content link to related entity data to a service consumer, which allows for simpler and more dynamic composition logic. The service consumer effectively invokes the relevant service through the returned link to obtain the related entity data.
* The Legacy Wrapper pattern can be applied to create a service to expose the legacy system through a standardized service contract. The core logic of the wrapping service would provide all necessary data transformation functionality to convert between inventory-standardized data representations and the proprietary format. The Endpoint Redirection pattern can be applied to return a link to related entity data to a service consumer, which allows for simpler and more dynamic composition logic. The service consumer effectively invokes the relevant service through the returned link to obtain the related entity data.
The Lightweight Endpoint pattern can be applied to establish lightweight capabilities that can return related entity data directly to service consumers, simplifying the composition logic by minimizing the number of services required to support simple business functions like order management or bill payment. This approach provides a standardized and simplified interface for the legacy system, reducing the complexity of the integration process with the entity services, and enabling them to focus on their core functionality.

QUESTION 20



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.

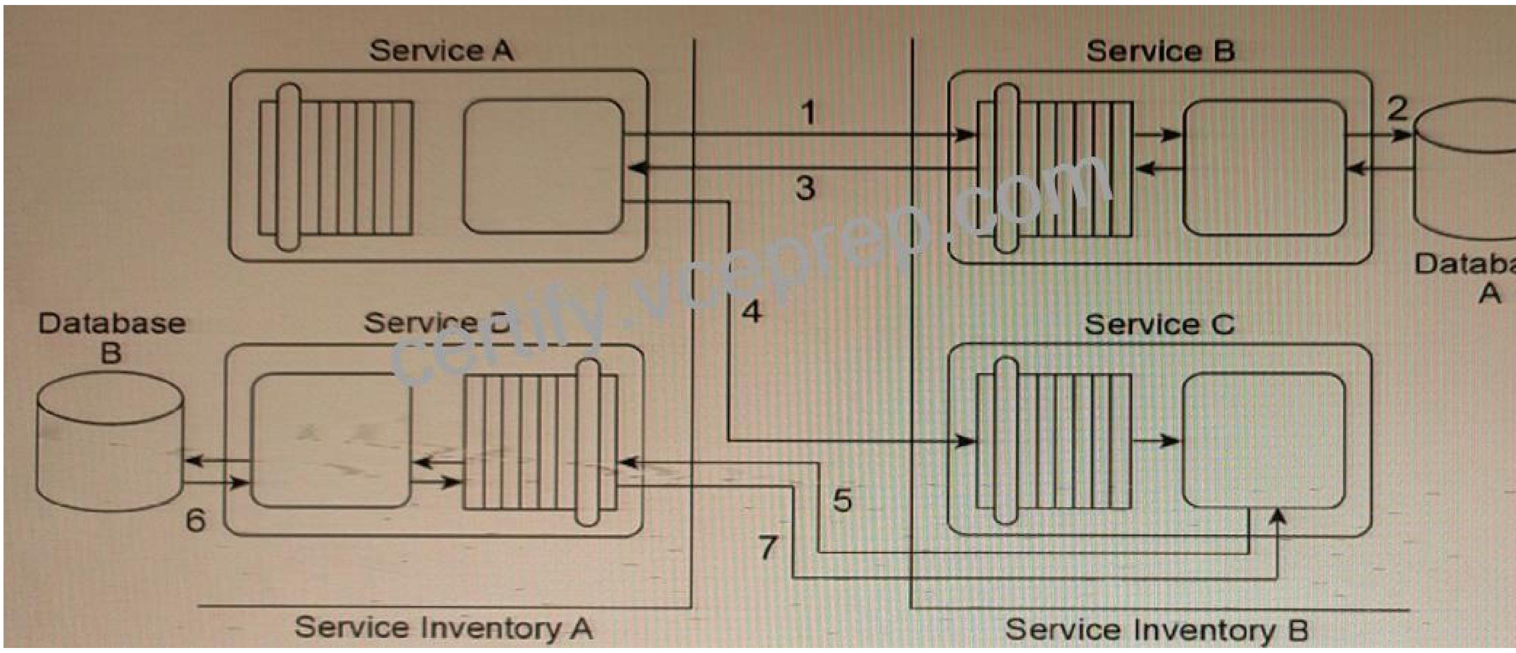
The current service composition architecture is using up too many resources due to the repeated invocation of Service A by Service Consumer A and the resulting message exchanges that occur with each invocation.

What steps can be taken to solve this problem?
* 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.
* 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.
* 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.
* 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.
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.

QUESTION 21



Service A sends a message to Service B (1). After Service B writes the message contents to Database A (2), it issues a response message back to Service A (3). Service A then sends a message to Service C (4). Upon receiving this message, Service C sends a message to Service D (5), which then writes the message contents to Database B (6) and issues a response message back to Service C (7).

Service A and Service D are located in Service Inventory A. Service B and Service C are located in Service Inventory B.

You are told that In this service composition architecture, all four services are exchanging invoice-related data in an XML format. However, the services in Service Inventory A are standardized to use a different XML schema for invoice data than the services in Service Inventory B. Also, Database A can only accept data in the Comma Separated Value (CSV) format and therefore cannot accept XML-formatted data. Database B only accepts XML-formatted data. However, it is a legacy database that uses a proprietary XML schema to represent invoice data that is different from the XML schema used by services in Service Inventory A or Service Inventory B.

What steps can be taken to enable the planned data exchange between these four services?
* The Data Model Transformation pattern can be applied so that data model transformation logic is positioned between Service A and Service B, between Service C and Service D, and between the Service D logic and Database B. The Data Format Transformation pattern can be applied so that data format transformation logic is positioned between Service A and Service C, and between the Service B logic and Database A.
* The Protocol Bridging pattern can be applied so that protocol conversion logic is positioned between the Service B logic and Database A. The Data Format Transformation pattern can be applied so that data format transformation logic is positioned between Service A and Service B, between Service A and Service C, between Service C and Service D, and between the Service D logic and Database B.
* The Data Model Transformation pattern can be applied so that data model transformation logic is positioned between Service A and Service B, between Service A and Service C, between Service C andService D, and between the Service D logic and Database B. The Data Format Transformation pattern can be applied so that data format transformation logic is positioned between the Service B logic and Database A.
* The Protocol Bridging pattern can be applied so that protocol conversion logic is positioned between Service A and Service B, between Service A and Service C, and between Service C and Service D. The Data Format Transformation pattern can be applied so

that data format transformation logic is positioned between the Service B logic and Database A and between the Service D logic and Database B
Explanation

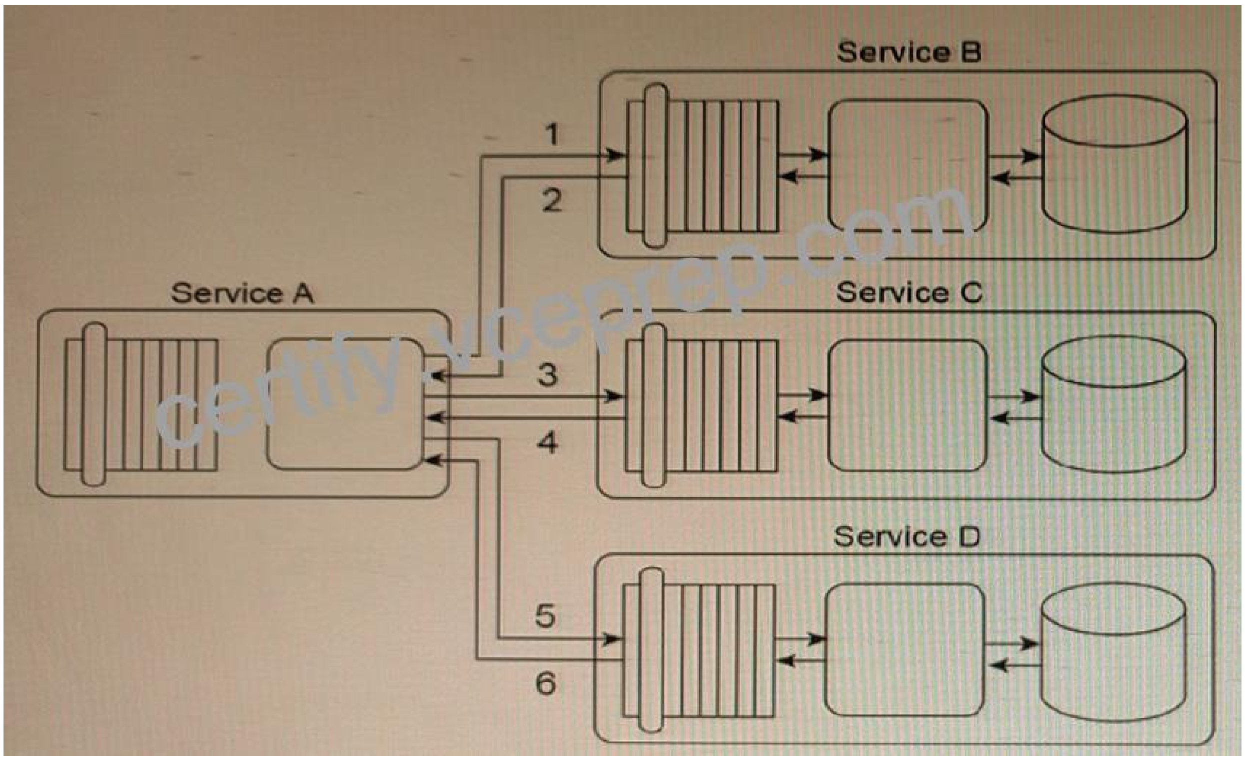
This solution addresses the two main challenges in the service composition architecture: the different XML schema used by services in Service Inventory A and Service Inventory B, and the incompatible data formats of the two databases.

By applying the Data Model Transformation pattern, data model transformation logic can be inserted to map the invoice-related data between the different XML schemas used by the services in Service Inventory A and Service Inventory B. This can be done at the appropriate points in the message flow: between Service A and Service B, between Service A and Service C, between Service C and Service D, and between the Service D logic and Database B.

By applying the Data Format Transformation pattern, data format transformation logic can be inserted to convert the XML-formatted data used by the services to the CSV format required by Database A, and to convert the proprietary XML schema used by Database B to the XML schema used by the services. This can be done between the Service B logic and Database A.

The Protocol Bridging pattern is not necessary in this case because all services are already communicating using the same protocol (presumably HTTP or a similar protocol).

QUESTION 22



Service A is a task service that is required to carry out a series of updates to a set of databases in order to complete a task. To perform the database updates. Service A must interact with three other services that each provides standardized data access capabilities.

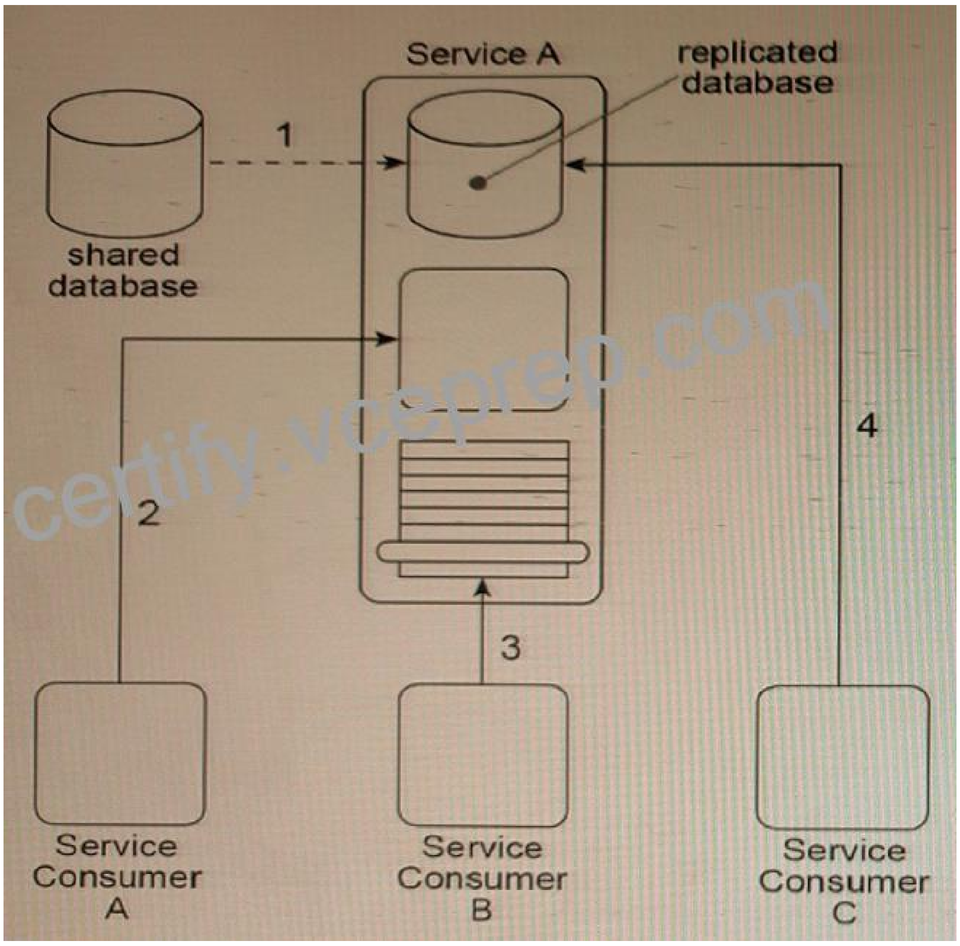
Service A sends its first update request message to Service B (1), which then responds with a message containing either a success or failure code (2). Service A then sends its second update request message to Service C (3), which also responds with a message containing either a success or failure code (4). Finally, Service A sends a request message to Service D (5), which responds with its own message containing either a success or failure code (6).

Services B, C and D are agnostic services that are reused and shared by multiple service consumers. This has caused unacceptable performance degradation for the service consumers of Service A as it is taking too long to complete its overall task. You&#8217;ve been asked to enhance the service composition architecture so that Service A provides consistent and predictable runtime performance. You are furthermore notified that a new type of data will be introduced to all three databases. It is important that this data is exchanged in a standardized manner so that the data model used for the data in inter-service messages is the same.

What steps can be taken to fulfill these requirements?
* The Compensating Service Transaction pattern can be applied so that exception logic is executed to notify Service A whenever the data access logic executed by Service B, C, or D takes too long. If the execution time exceeds a predefined limit, then the overall service activity is cancelled and a failure code is returned to Service A. The Schema Centralization pattern is applied to ensure that all services involved in the composition use the same schemas to represented the data consistently.
* The Composition Autonomy pattern can be applied to establish an isolated environment in which redundant implementations of Services B, C and D are accessed only by Service A. The Canonical Schema pattern can be applied to ensure that the new type of data is represented by the same data model, regardless of which service sends or receives a message containing the data.
* The Redundant Implementation pattern is applied to Service A, along with the Service Instance Routing pattern. This allows for multiple instances of Service A to be created across multiple physical implementations, thereby increasing scalability and availability. The Dual Protocols pattern is applied to all services to support proprietary and standardized data models.
* The Service Fagade pattern is applied to all services in order to create an intermediary processing layer within each service architecture. The Content Negotiation pattern is applied so that each service fagade component within each service architecture is equipped with the logic required to defer request messages to other service instances when concurrent usage of the service is high, and to further apply the conversation logic necessary to convert proprietary data from a database into the standardized XML schema format.
Explanation

This approach isolates the services used by Service A, allowing it to avoid the performance degradation caused by multiple service consumers. By creating redundant implementations of Services B, C, and D that are accessed only by Service A, the Composition Autonomy pattern also ensures that Service A&#8217;s runtime performance is consistent and predictable. Applying the Canonical Schema pattern ensures that the new type of data is exchanged in a standardized manner, ensuring consistent representation of the data model used for the data in inter-service messages.

QUESTION 23



Service A is a utility service that provides generic data access logic to a database containing data that is periodically replicated from a shared database (1). Because the Standardized Service Contract principle was applied to the design of Service A, its service contract has been fully standardized.

The service architecture of Service A Is being accessed by three service consumers. Service Consumer A accesses a component that is partof the Service A Implementation by Invoking it directly (2). Service Consumer B invokes Service A by accessing Its service contract (3). Service Consumer C directly accesses the replicated database that Is part of the Service A Implementation (4).

You&#8217;ve been told that the reason Service Consumers A and C bypass the published Service A service contract is because, for security reasons, they are not allowed to access a subset of the capabilities inthe API that comprises the Service A service contract. How can the Service A architecture be changed to enforce these security restrictions while avoiding negative forms of coupling?
* The Contract Centralization pattern can be applied to force all service consumers to access the Service A architecture via its published service contract. This will prevent negative forms of coupling that could lead to problems when the database is replaced. The Service Abstraction principle can then be applied to hide underlying service architecture details so that future service consumers cannot be designed to access any part of the underlying service implementation.
* The Contract Centralization pattern can be applied to force service consumers to access the Service A architecture via its published service contract only. The Service Loose Coupling principle can then be applied to ensure that the centralized service contract does not contain any content that is dependent on or derived from the underlying service implementation.
* The Contract Centralization pattern can be applied to force service consumers to access the Service A architecture via its published service contract only. The Concurrent Contracts pattern can be applied to Service A in order to establish one or more alternative service contracts. This allows service consumers with different levels of authorization to access different types of service logic via Service A&#8217;s published service contracts.

* The Contract Centralization pattern can be applied to force service consumers to access the Service A architecture via its published service contract only. The Idempotent Capability pattern can be applied to Service A to establish alternative sets of service capabilities for service consumers with different levels of authorization.
Explanation

The Contract Centralization pattern can be applied to force service consumers to access the Service A architecture via its published service contract only. The Service Loose Coupling principle can then be applied to ensure that the centralized service contract does not contain any content that is dependent on or derived from the underlying service implementation. This will enforce the security restrictions while avoiding negative forms of coupling. By ensuring loose coupling, changes to the implementation of Service A will not require changes to its published service contract, making it easier to maintain and evolve the service.

SOA S90.08B exam is designed to test the candidate's knowledge and skills in designing and implementing SOA solutions using services and microservices. S90.08B exam consists of multiple-choice questions and requires the candidate to demonstrate their ability to apply SOA design and architecture principles to real-world problems.