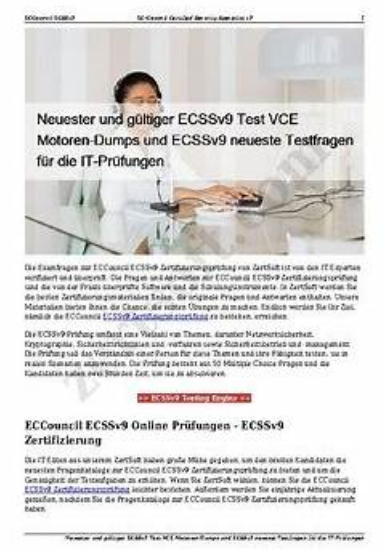
Neuester und gültiger ACD301 Test VCE Motoren-Dumps und ACD301 neueste Testfragen für die IT-Prüfungen



Übrigens, Sie können die vollständige Version der EchteFrage ACD301 Prüfungsfragen aus dem Cloud-Speicher herunterladen: https://drive.google.com/open?id=14mXS0ZGG5qZqIjhl6AYqnY-i7gtPlFvK

Die Fragenpool zur Appian ACD301 Zertifizierungsprüfung von EchteFrage hat eine große Ähnlichkeit mit den realen Prüfungen. Sie können in unseren Fragenpool den realen Prüfungsfragen begegnen. Das zeigt die Fähigkeiten unseres Expertenteams. Nun sind viele IT-Fachleute ganz ambitioniert. Sie beteiligen sich an der Appian ACD301 Zertifizierungsprüfung, um sich den Bedürfnissen des Marktes anzupassen und ihren Traum zu verwirklichen.

Es existiert viele Methoden, mit der Sie sich auf die Appian ACD301 Zertifizierungsprüfung vorzubereiten. Unsere Website bietet zuverlässige Prüfungsmaterialien, mit den Sie sich auf die nächste Appian ACD301 Zertifizierungsprüfung vorbereiten. Die Lernmaterialien zur Appian ACD301 Zertifizierungsprüfung von EchteFrage enthalten sowohl Fragen als auch Antworten. Unsere Materialien sind von der Praxis überprüfte Software. Wir werden alle Ihren Bedürfnisse zurAppian ACD301 Zertifizierung abdecken.

>> ACD301 PDF Demo <<

Die neuesten ACD301 echte Prüfungsfragen, Appian ACD301 originale

fragen

Nun ist eine Gesellschaft, die mit den fähigen Leuten überschwemmt. Aber viele Fachleute fehlen trotzdem doch. Beispielsweise fehlen in der IT-Branche Techniker. Und die Appian ACD301 Zertifizierungsprüfung sit eine Prüfung, die IT-Technik testet. EchteFrage ist eine Website, die Ihnen Kenntnise zur Appian ACD301 Zertifizierungsprüfung liefert.

Appian ACD301 Prüfungsplan:

Thema	Einzelheiten
Thema 1	• Platform Management: This section of the exam measures skills of Appian System Administrators and covers the ability to manage platform operations such as deploying applications across environments, troubleshooting platform-level issues, configuring environment settings, and understanding platform architecture. Candidates are also expected to know when to involve Appian Support and how to adjust admin console configurations to maintain stability and performance.
Thema 2	• Project and Resource Management: This section of the exam measures skills of Agile Project Leads and covers interpreting business requirements, recommending design options, and leading Agile teams through technical delivery. It also involves governance, and process standardization.
Thema 3	• Proactively Design for Scalability and Performance: This section of the exam measures skills of Application Performance Engineers and covers building scalable applications and optimizing Appian components for performance. It includes planning load testing, diagnosing performance issues at the application level, and designing systems that can grow efficiently without sacrificing reliability.

Appian Lead Developer ACD301 Prüfungsfragen mit Lösungen (Q39-Q44):

39. Frage
You are designing a process that is anticipated to be executed multiple times a day. This process retrieves data from an external system and then calls various utility processes as needed. The main process will not use the results of the utility processes, and there are no user forms anywhere.
Which design choice should be used to start the utility processes and minimize the load on the execution engines?

- A. Start the utility processes via a subprocess asynchronously.
- B. Use Process Messaging to start the utility process.
- C. Use the Start Process Smart Service to start the utility processes.
- D. Start the utility processes via a subprocess synchronously.

Antwort: A

Begründung:
Comprehensive and Detailed In-Depth Explanation:As an Appian Lead Developer, designing a process that executes frequently (multiple times a day) and calls utility processes without using their results requires optimizing performance and minimizing load on Appian's execution engines. The absence of user forms indicates a backend process, so user experience isn't a concern-only engine efficiency matters. Let's evaluate each option:
* A. Use the Start Process Smart Service to start the utility processes:The Start Process Smart Service launches a new process instance independently, creating a separate process in the Work Queue. While functional, it increases engine load because each utility process runs as a distinct instance, consuming engine resources and potentially clogging the Java Work Queue, especially with frequent executions.
Appian's performance guidelines discourage unnecessary separate process instances for utility tasks, favoring integrated subprocesses, making this less optimal.
* B. Start the utility processes via a subprocess synchronously:Synchronous subprocesses (e.g., a!
startProcess with isAsync: false) execute within the main process flow, blocking until completion. For utility processes not used by the main process, this creates unnecessary delays, increasing execution time and engine load. With frequent daily executions, synchronous subprocesses could strain engines, especially if utility processes are slow or numerous. Appian's documentation recommends asynchronous execution for non-dependent, non-blocking tasks, ruling this out.
* C. Use Process Messaging to start the utility process:Process Messaging (e.g., sendMessage() in Appian) is used for inter-process communication, not for starting processes. It's designed to pass data between running processes, not initiate new ones. Attempting to use it for starting utility processes would require additional setup (e.g., a listening process) and isn't a standard or efficient method.

Appian's messaging features are for coordination, not process initiation, making this inappropriate.
* D. Start the utility processes via a subprocess asynchronously:This is the best choice. Asynchronous subprocesses (e.g., a!startProcess with isAsync: true) execute independently of the main process, offloading work to the engine without blocking or delaying the parent process. Since the main process doesn't use the utility process results and there are no user forms, asynchronous execution minimizes engine load by distributing tasks across time, reducing Work Queue pressure during frequent executions. Appian's performance best practices recommend asynchronous subprocesses for non- dependent, utility tasks to optimize engine utilization, making this ideal for minimizing load.
Conclusion: Starting the utility processes via a subprocess asynchronously (D) minimizes engine load by allowing independent execution without blocking the main process, aligning with Appian's performance optimization strategies for frequent, backend processes.
References:
* Appian Documentation: "Process Model Performance" (Synchronous vs. Asynchronous Subprocesses).
* Appian Lead Developer Certification: Process Design Module (Optimizing Engine Load).
* Appian Best Practices: "Designing Efficient Utility Processes" (Asynchronous Execution).

40. Frage
Your Appian project just went live with the following environment setup: DEV > TEST (SIT/UAT) > PROD.
Your client is considering adding a support team to manage production defects and minor enhancements, while the original development team focuses on Phase 2. Your client is asking you for a new environment strategy that will have the least impact on Phase 2 development work. Which optioninvolves the lowest additional server cost and the least code retrofit effort?

- A. Phase 2 development work stream: DEV > TEST (SIT/UAT) > PROD Production support work stream: DEV > TEST2 (SIT/UAT) > PROD
- B. Phase 2 development work stream: DEV > TEST (SIT) > STAGE (UAT) > PROD Production support work stream: DEV > TEST2 (SIT/UAT) > PROD
- C. Phase 2 development work stream: DEV > TEST (SIT) > STAGE (UAT) > PROD Production support work stream: DEV2 > STAGE (SIT/UAT) > PROD
- D. Phase 2 development work stream: DEV > TEST (SIT/UAT) > PROD Production support work stream: DEV2 > TEST (SIT/UAT) > PROD

Antwort: A

Begründung:
Comprehensive and Detailed In-Depth Explanation:The goal is to design an environment strategy that minimizes additional server costs and code retrofit effort while allowing the support team to manage production defects and minor enhancements without disrupting the Phase 2 development team. The current setup (DEV > TEST (SIT/UAT) > PROD) uses a single development and testing pipeline, and the client wants to segregate support activities from Phase 2 development. Appian's Environment Management Best Practices emphasize scalability, cost efficiency, and minimal refactoring when adjusting environments.
* Option C (Phase 2 development work stream: DEV > TEST (SIT/UAT) > PROD; Production support work stream: DEV > TEST2 (SIT/UAT) > PROD):This option is the most cost-effective and requires the least code retrofit effort. It leverages the existing DEV environment for both teams but introduces a separate TEST2 environment for the support team's SIT/UAT activities. Since DEV is already shared, no new development server is needed, minimizing server costs. The existing code in DEV and TEST can be reused for TEST2 by exporting and importing packages, with minimal adjustments (e.g., updating environment-specific configurations). The Phase 2 team continues using the original TEST environment, avoiding disruption. Appian supports multiple test environments branching from a single DEV, and the PROD environment remains shared, aligning with the client's goal of low impact on Phase 2. The support team can handle defects and enhancements in TEST2 without interfering with development workflows.
* Option A (Phase 2 development work stream: DEV > TEST (SIT) > STAGE (UAT) > PROD; Production support work stream: DEV > TEST2 (SIT/UAT) > PROD):This introduces a STAGE environment for UAT in the Phase 2 stream, adding complexity and potentially requiring code updates to accommodate the new environment (e.g., adjusting deployment scripts). It also requires a new TEST2 server, increasing costs compared to Option C, where TEST2 reuses existing infrastructure.
* Option B (Phase 2 development work stream: DEV > TEST (SIT) > STAGE (UAT) > PROD; Production support work stream: DEV2 > STAGE (SIT/UAT) > PROD):This option adds both a DEV2 server for the support team and a STAGE environment, significantly increasing server costs. It also requires refactoring code to support two development environments (DEV and DEV2), including duplicating or synchronizing objects, which is more effort than reusing a single DEV.
* Option D (Phase 2 development work stream: DEV > TEST (SIT/UAT) > PROD; Production support work stream: DEV2 > TEST (SIT/UAT) > PROD):This introduces a DEV2 server for the support team, adding server costs. Sharing the TEST environment between teams could lead to conflicts (e.g., overwriting test data), potentially disrupting Phase 2 development. Code retrofit effort is higher due to managing two DEV environments and ensuring TEST compatibility.
Cost and Retrofit Analysis:
* Server Cost:Option C avoids new DEV or STAGE servers, using only an additional TEST2, which can often be provisioned on

existing hardware or cloud resources with minimal cost. Options A, B, and D require additional servers (TEST2, DEV2, or STAGE), increasing expenses.
* Code Retrofit:Option C minimizes changes by reusing DEV and PROD, with TEST2 as a simple extension. Options A and B require updates for STAGE, and B and D involve managing multiple DEV environments, necessitating more significant refactoring.
Appian's recommendation for environment strategies in such scenarios is to maximize reuse of existing infrastructure and avoid unnecessary environment proliferation, making Option C the optimal choice.
References:Appian Documentation - Environment Management and Deployment, Appian Lead Developer Training - Environment Strategy and Cost Optimization.

41. Frage
You are required to configure a connection so that Jira can inform Appian when specific tickets change (using a webhook). Which three required steps will allow you to connect both systems?

- A. Create a new API Key and associate a service account.
- B. Create an integration object from Appian to Jira to periodically check the ticket status.
- C. Give the service account system administrator privileges.
- D. Configure the connection in Jira specifying the URL and credentials.
- E. Create a Web API object and set up the correct security.

Antwort: A,D,E

Begründung:
Comprehensive and Detailed In-Depth Explanation:
Configuring a webhook connection from Jira to Appian requires setting up a mechanism for Jira to push ticket change notifications to Appian in real-time. This involves creating an endpoint in Appian to receive the webhook and configuring Jira to send the data. Appian's Integration Best Practices and Web API documentation provide the framework for this process.
Option A (Create a Web API object and set up the correct security):
This is a required step. In Appian, a Web API object serves as the endpoint to receive incoming webhook requests from Jira. You must define the API structure (e.g., HTTP method, input parameters) and configure security (e.g., basic authentication, API key, or OAuth) to validate incoming requests. Appian recommends using a service account with appropriate permissions to ensure secure access, aligning with the need for a controlled webhook receiver.
Option B (Configure the connection in Jira specifying the URL and credentials):
This is essential. In Jira, you need to set up a webhook by providing the Appian Web API's URL (e.g., https://<appian-site>/suite/webapi/<web-api-name>) and the credentials or authentication method (e.g., API key or basic auth) that match the security setup in Appian. This ensures Jira can successfully send ticket change events to Appian.
Option C (Create a new API Key and associate a service account):
This is necessary for secure authentication. Appian recommends using an API key tied to a service account for webhook integrations. The service account should have permissions to process the incoming data (e.g., write to a process or data store) but not excessive privileges. This step complements the Web API security setup and Jira configuration.
Option D (Give the service account system administrator privileges):
This is unnecessary and insecure. System administrator privileges grant broad access, which is overkill for a webhook integration. Appian's security best practices advocate for least-privilege principles, limiting the service account to the specific objects or actions needed (e.g., executing the Web API).
Option E (Create an integration object from Appian to Jira to periodically check the ticket status):
This is incorrect for a webhook scenario. Webhooks are push-based, where Jira notifies Appian of changes. Creating an integration object for periodic polling (pull-based) is a different approach and not required for the stated requirement of Jira informing Appian via webhook.
These three steps (A, B, C) establish a secure, functional webhook connection without introducing unnecessary complexity or security risks.
Reference:
The three required steps that will allow you to connect both systems are:
A . Create a Web API object and set up the correct security. This will allow you to define an endpoint in Appian that can receive requests from Jira via webhook. You will also need to configure the security settings for the Web API object, such as authentication method, allowed origins, and access control.
B . Configure the connection in Jira specifying the URL and credentials. This will allow you to set up a webhook in Jira that can send requests to Appian when specific tickets change. You will need to specify the URL of the Web API object in Appian, as well as any credentials required for authentication.
C . Create a new API Key and associate a service account. This will allow you to generate a unique token that can be used for authentication between Jira and Appian. You will also need to create a service account in Appian that has permissions to access or update data related to Jira tickets.

The other options are incorrect for the following reasons:
D . Give the service account system administrator privileges. This is not required and could pose a security risk, as giving system administrator privileges to a service account could allow it to perform actions that are not related to Jira tickets, such as modifying system settings or accessing sensitive data.
E . Create an integration object from Appian to Jira to periodically check the ticket status. This is not required and could cause unnecessary overhead, as creating an integration object from Appian to Jira would involve polling Jira for ticket status changes, which could consume more resources than using webhook notifications. Verified Reference: Appian Documentation, section "Web API" and "API Keys".

42. Frage
An Appian application contains an integration used to send a JSON, called at the end of a form submission, returning the created code of the user request as the response. To be able to efficiently follow their case, the user needs to be informed of that code at the end of the process. The JSON contains case fields (such as text, dates, and numeric fields) to a customer's API. What should be your two primary considerations when building this integration?

- A. The size limit of the body needs to be carefully followed to avoid an error.
- B. A process must be built to retrieve the API response afterwards so that the user experience is not impacted.
- C. A dictionary that matches the expected request body must be manually constructed.
- D. The request must be a multi-part POST.

Antwort: A,C

Begründung:
Comprehensive and Detailed In-Depth Explanation:
As an Appian Lead Developer, building an integration to send JSON to a customer's API and return a code to the user involves balancing usability, performance, and reliability. The integration is triggered at form submission, and the user must see the response (case code) efficiently. The JSON includes standard fields (text, dates, numbers), and the focus is on primary considerations for the integration itself. Let's evaluate each option based on Appian's official documentation and best practices:
A . A process must be built to retrieve the API response afterwards so that the user experience is not impacted:
This suggests making the integration asynchronous by calling it in a process model (e.g., via a Start Process smart service) and retrieving the response later, avoiding delays in the UI. While this improves user experience for slow APIs (e.g., by showing a "Processing" message), it contradicts the requirement that the user is "informed of that code at the end of the process." Asynchronous processing would delay the code display, requiring additional steps (e.g., a follow-up task), which isn't efficient for this use case. Appian's default integration pattern (synchronous call in an Integration object) is suitable unless latency is a known issue, making this a secondary-not primary-consideration.
B . The request must be a multi-part POST:
A multi-part POST (e.g., multipart/form-data) is used for sending mixed content, like files and text, in a single request. Here, the payload is a JSON containing case fields (text, dates, numbers)-no files are mentioned. Appian's HTTP Connected System and Integration objects default to application/json for JSON payloads via a standard POST, which aligns with REST API norms. Forcing a multi-part POST adds unnecessary complexity and is incompatible with most APIs expecting JSON. Appian documentation confirms this isn't required for JSON-only data, ruling it out as a primary consideration.
C . The size limit of the body needs to be carefully followed to avoid an error:
This is a primary consideration. Appian's Integration object has a payload size limit (approximately 10 MB, though exact limits depend on the environment and API), and exceeding it causes errors (e.g., 413 Payload Too Large). The JSON includes multiple case fields, and while "hundreds of thousands" isn't specified, large datasets could approach this limit. Additionally, the customer's API may impose its own size restrictions (common in REST APIs). Appian Lead Developer training emphasizes validating payload size during design-e.g., testing with maximum expected data-to prevent runtime failures. This ensures reliability and is critical for production success.
D . A dictionary that matches the expected request body must be manually constructed:
This is also a primary consideration. The integration sends a JSON payload to the customer's API, which expects a specific structure (e.g., { "field1": "text", "field2": "date" }). In Appian, the Integration object requires a dictionary (key-value pairs) to construct the JSON body, manually built to match the API's schema. Mismatches (e.g., wrong field names, types) cause errors (e.g., 400 Bad Request) or silent failures. Appian's documentation stresses defining the request body accurately-e.g., mapping form data to a CDT or dictionary-ensuring the API accepts the payload and returns the case code correctly. This is foundational to the integration's functionality.
Conclusion: The two primary considerations are C (size limit of the body) and D (constructing a matching dictionary). These ensure the integration works reliably (C) and meets the API's expectations (D), directly enabling the user to receive the case code at submission end. Size limits prevent technical failures, while the dictionary ensures data integrity-both are critical for a synchronous JSON POST in Appian. Option A could be relevant for performance but isn't primary given the requirement, and B is irrelevant to the scenario.

Reference:
Appian Documentation: "Integration Object" (Request Body Configuration and Size Limits).
Appian Lead Developer Certification: Integration Module (Building REST API Integrations).
Appian Best Practices: "Designing Reliable Integrations" (Payload Validation and Error Handling).

43. Frage
You have an active development team (Team A) building enhancements for an application (App X) and are currently using the TEST environment for User Acceptance Testing (UAT).
A separate operations team (Team B) discovers a critical error in the Production instance of App X that they must remediate. However, Team B does not have a hotfix stream for which to accomplish this. The available environments are DEV, TEST, and PROD.
Which risk mitigation effort should both teams employ to ensure Team A's capital project is only minorly interrupted, and Team B's critical fix can be completed and deployed quickly to end users?

- A. Team B must address changes in the TEST environment. These changes can then be tested and deployed directly to PROD. Once the deployment is complete, Team B can then communicate their changes to Team A to ensure they are incorporated as part of the next release.
- B. Team A must analyze their current codebase in DEV to merge the hotfix changes into their latest enhancements. Team B is then required to wait for the hotfix to follow regular deployment protocols from DEV to the PROD environment.
- C. Team B must communicate to Team A which component will be addressed in the hotfix to avoid overlap of changes. If overlap exists, the component must be versioned to its PROD state before being remediated and deployed, and then versioned back to its latest development state. If overlap does not exist, the component may be remediated and deployed without any version changes.
- D. Team B must address the changes directly in PROD. As there is no hotfix stream, and DEV and TEST are being utilized for active development, it is best to avoid a conflict of components. Once Team A has completed their enhancements work, Team B can update DEV and TEST accordingly.

Antwort: C

Begründung:
Comprehensive and Detailed In-Depth Explanation:
As an Appian Lead Developer, managing concurrent development and operations (hotfix) activities across limited environments (DEV, TEST, PROD) requires minimizing disruption to Team A's enhancements while ensuring Team B's critical fix reaches PROD quickly. The scenario highlights no hotfix stream, active UAT in TEST, and a critical PROD issue, necessitating a strategic approach.
Let's evaluate each option:
A . Team B must communicate to Team A which component will be addressed in the hotfix to avoid overlap of changes. If overlap exists, the component must be versioned to its PROD state before being remediated and deployed, and then versioned back to its latest development state. If overlap does not exist, the component may be remediated and deployed without any version changes:
This is the best approach. It ensures collaboration between teams to prevent conflicts, leveraging Appian's version control (e.g., object versioning in Appian Designer). Team B identifies the critical component, checks for overlap with Team A's work, and uses versioning to isolate changes. If no overlap exists, the hotfix deploys directly; if overlap occurs, versioning preserves Team A's work, allowing the hotfix to deploy and then reverting the component for Team A's continuation. This minimizes interruption to Team A's UAT, enables rapid PROD deployment, and aligns with Appian's change management best practices.
B . Team A must analyze their current codebase in DEV to merge the hotfix changes into their latest enhancements. Team B is then required to wait for the hotfix to follow regular deployment protocols from DEV to the PROD environment:
This delays Team B's critical fix, as regular deployment (DEV → TEST → PROD) could take weeks, violating the need for "quick deployment to end users." It also risks introducing Team A's untested enhancements into the hotfix, potentially destabilizing PROD. Appian's documentation discourages mixing development and hotfix workflows, favoring isolated changes for urgent fixes, making this inefficient and risky.
C . Team B must address changes in the TEST environment. These changes can then be tested and deployed directly to PROD. Once the deployment is complete, Team B can then communicate their changes to Team A to ensure they are incorporated as part of the next release:
Using TEST for hotfix development disrupts Team A's UAT, as TEST is already in use for their enhancements. Direct deployment from TEST to PROD skips DEV validation, increasing risk, and doesn't address overlap with Team A's work. Appian's deployment guidelines emphasize separate streams (e.g., hotfix streams) to avoid such conflicts, making this disruptive and unsafe.
D . Team B must address the changes directly in PROD. As there is no hotfix stream, and DEV and TEST are being utilized for active development, it is best to avoid a conflict of components. Once Team A has completed their enhancements work, Team B can update DEV and TEST accordingly:
Making changes directly in PROD is highly discouraged in Appian due to lack of testing, version control, and rollback capabilities, risking further instability. This violates Appian's Production governance and security policies, and delays Team B's updates until

Team A finishes, contradicting the need for a "quick deployment." Appian's best practices mandate using lower environments for changes, ruling this out.
Conclusion: Team B communicating with Team A, versioning components if needed, and deploying the hotfix (A) is the risk mitigation effort. It ensures minimal interruption to Team A's work, rapid PROD deployment for Team B's fix, and leverages Appian's versioning for safe, controlled changes-aligning with Lead Developer standards for multi-team coordination.
Reference:
Appian Documentation: "Managing Production Hotfixes" (Versioning and Change Management).
Appian Lead Developer Certification: Application Management Module (Hotfix Strategies).
Appian Best Practices: "Concurrent Development and Operations" (Minimizing Risk in Limited Environments).

44. Frage

.....

Die Zertifizierungsantworten zur Appian ACD301 Zertifizierungsprüfung von EchteFrage sind die Grundbedarfsgüter der Kandidaten, mit deren Sie sich ausreichend auf die Appian ACD301 Prüfung vorbereiten und selbstsicherer die Prüfung machen können. Sie sind seht zielgerichtet und von guter Qualität. Nur EchteFrage könnte so perfekt sein.