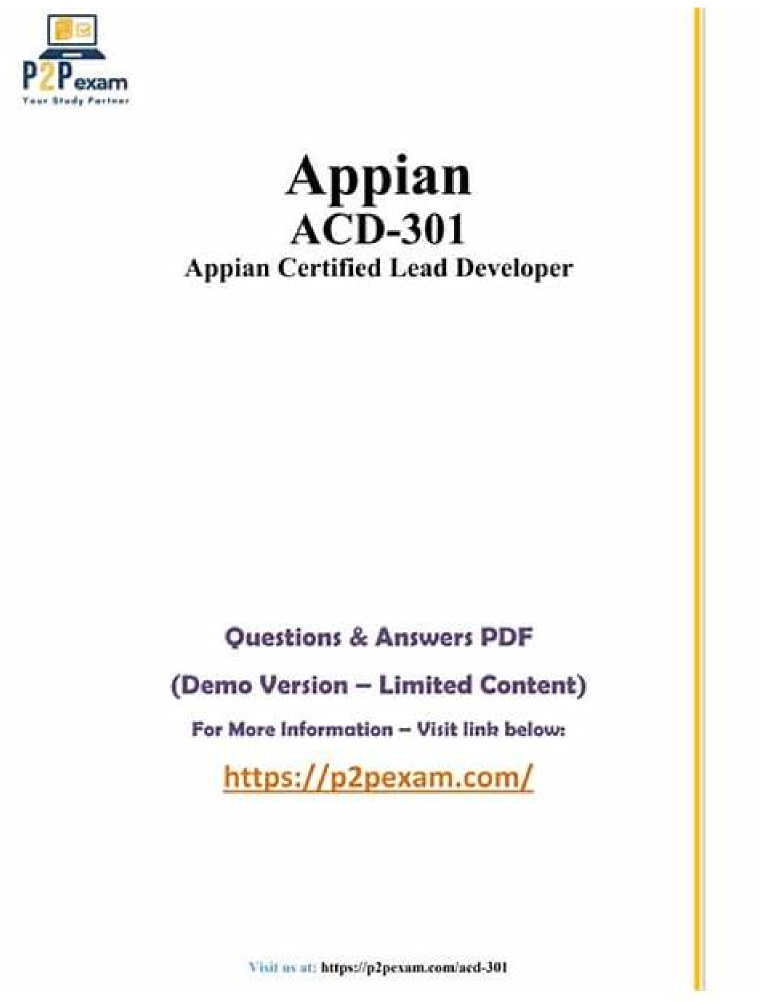
ACD301資格トレーリング & ACD301復習テキスト



P2Pexam
Your Study Partner

Appian
ACD-301
Appian Certified Lead Developer

Questions & Answers PDF
(Demo Version – Limited Content)
For More Information – Visit link below:
https://p2pexam.com/

Visit us at: https://p2pexam.com/acd-301

無料でクラウドストレージから最新のTech4Exam ACD301 PDFダンプをダウンロードする：https://drive.google.com/open?id=1NDwiSTb15zI_GuEOtnLX7xM37CtWmz-L

AppianのACD301認定試験は現在で本当に人気がある試験ですね。まだこの試験の認定資格を取っていないあなたも試験を受ける予定があるのでしょうか。確かに、これは困難な試験です。しかし、難しいといっても、高い点数を取って楽に試験に合格できないというわけではないです。では、まだ試験に合格するショートカットがわからないあなたは、受験のテクニックを知りたいですか。今教えてあげますよ。Tech4ExamのACD301問題集を利用することです。

Tech4Examは客様の要求を満たせていい評判をうけいたします。たくさんのひとは弊社の商品を使って、試験に順調に合格しました。そして、かれたちがリピーターになりました。Tech4Examが提供したAppianのACD301試験問題と解答が真実の試験の練習問題と解答は最高の相似性があり、一年の無料オンラインの更新のサービスがあり、100%のパス率を保証して、もし試験に合格しないと、弊社は全額で返金いたします。

>> ACD301資格トレーリング <<

信頼できるACD301資格トレーリング & 合格スムーズACD301復習テキスト | 実用的なACD301関連日本語版問題集

Tech4Examが提供しておりますのは専門家チームの研究したACD301問題と真題で弊社の高い名誉はたぶり信頼をうけられます。安心で弊社の商品を使うために無料なACD301サンプルをダウンロードしてください。

Appian ACD301 認定試験の出題範囲：

トピック	出題範囲
トピック 1	• プラットフォーム管理：このセクションでは、Appian システム管理者のスキルを評価します。環境間でのアプリケーションの展開、プラットフォームレベルの問題のトラブルシューティング、環境設定の構成、プラットフォームアーキテクチャの理解など、プラットフォーム運用の管理能力が問われます。受験者は、Appian サポートをいつ呼び出すべきか、また、安定性とパフォーマンスを維持するために管理コンソールの設定を調整する方法も理解していることが求められます。
トピック 2	• スケーラビリティとパフォーマンスを積極的に設計：この試験セクションでは、アプリケーションパフォーマンスエンジニアのスキルを評価し、スケーラブルなアプリケーションの構築とAppianコンポーネントのパフォーマンス最適化について学びます。負荷テストの計画、アプリケーションレベルでのパフォーマンス問題の診断、信頼性を犠牲にすることなく効率的に拡張できるシステムの設計などが含まれます。
トピック 3	• アプリケーション設計と開発：この試験セクションでは、リードAppian開発者のスキルを評価し、Appianの機能を活用してユーザーニーズを満たすアプリケーションの設計と開発について学びます。一貫性、再利用性、そしてチーム間の連携を考慮した設計も含まれます。複雑な環境で複数のスケーラブルなアプリケーションを構築するためのベストプラクティスの適用に重点が置かれます。
トピック 4	• データ管理：このセクションでは、データアーキテクトのスキルを評価し、データモデルの分析、設計、セキュリティ確保について学習します。受験者は、Appianのデータファブリックの使用方法とデータ移行の管理方法を理解していることを証明する必要があります。特に、大容量データ環境におけるパフォーマンスの確保、データ関連の問題の解決、高度なデータベース機能の効果的な実装に重点が置かれます。

Appian Lead Developer 認定 ACD301 試験問題 (Q38-Q43):

質問 # 38
While working on an application, you have identified oddities and breaks in some of your components. How can you guarantee that this mistake does not happen again in the future?

- A. Design and communicate a best practice that dictates designers only work within the confines of their own application.
- B. Create a best practice that enforces a peer review of the deletion of any components within the application.
- C. Ensure that the application administrator group only has designers from that application's team.
- D. Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and assign them the permissions to administer their application.

正解：B

解説：
Comprehensive and Detailed In-Depth Explanation:As an Appian Lead Developer, preventing recurring
"oddities and breaks" in application components requires addressing root causes-likely tied to human error, lack of oversight, or uncontrolled changes-while leveraging Appian's governance and collaboration features.
The question implies a past mistake (e.g., accidental deletions or modifications) and seeks a proactive, sustainable solution. Let's evaluate each option based on Appian's official documentation and best practices:
* A. Design and communicate a best practice that dictates designers only work within the confines of their own application:This suggests restricting designers to their assigned applications via a policy.
While Appian supports application-level security (e.g., Designer role scoped to specific applications), this approach relies on voluntary compliance rather than enforcement. It doesn't directly address
"oddities and breaks"-e.g., a designer could still mistakenly alter components within their own application. Appian's documentation emphasizes technical controls and process rigor over broad guidelines, making this insufficient as a guarantee.
* B. Ensure that the application administrator group only has designers from that application's team:This involves configuring security so only team-specific designers have Administrator rights to the application (via Appian's Security settings). While this limits external interference, it doesn't prevent internal mistakes (e.g., a team designer deleting a critical component). Appian's security model already restricts access by default, and the issue isn't about unauthorized access but rather component integrity.
This step is a hygiene factor, not a direct solution to the problem, and fails to "guarantee" prevention.
* C. Create a best practice that enforces a peer review of the deletion of any components within the application:This is the best

choice. A peer review process for deletions (e.g., process models, interfaces, or records) introduces a checkpoint to catch errors before they impact the application. In Appian, deletions are permanent and can cascade (e.g., breaking dependencies), aligning with the "oddities and breaks" described. While Appian doesn't natively enforce peer reviews, this can be implemented via team workflows-e.g., using Appian's collaboration tools (like Comments or Tasks) or integrating with version control practices during deployment. Appian Lead Developer training emphasizes change management and peer validation to maintain application stability, making this a robust, preventive measure that directly addresses the root cause.
* D. Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and assign them the permissions to administer their application:This option is confusingly worded but seems to suggest granting Designer system role permissions (a high-level privilege) while limiting developers to Viewer rights system-wide, withAdministrator rights only for their application. In Appian, the "Designer" system role grants broad platform access (e.g., creating applications), which contradicts "basic user rights" (Viewer role). Regardless, adjusting permissions doesn't prevent mistakes-it only controls who can make them. The issue isn't about access but about error prevention, so this option misses the mark and is impractical due to its contradictory setup.
Conclusion: Creating a best practice that enforces a peer review of the deletion of any components (C) is the strongest solution. It directly mitigates the risk of "oddities and breaks" by adding oversight to destructive actions, leveraging team collaboration, and aligning with Appian's recommended governance practices.
Implementation could involve documenting the process, training the team, and using Appian's monitoring tools (e.g., Application Properties history) to track changes-ensuring mistakes are caught before deployment.
This provides the closest guarantee to preventing recurrence.
References:
* Appian Documentation: "Application Security and Governance" (Change Management Best Practices).
* Appian Lead Developer Certification: Application Design Module (Preventing Errors through Process).
* Appian Best Practices: "Team Collaboration in Appian Development" (Peer Review Recommendations).

質問 # 39
While working on an application, you have identified oddities and breaks in some of your components. How can you guarantee that this mistake does not happen again in the future?

- A. Design and communicate a best practice that dictates designers only work within the confines of their own application.
- B. Create a best practice that enforces a peer review of the deletion of any components within the application.
- C. Ensure that the application administrator group only has designers from that application's team.
- D. Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and assign them the permissions to administer their application.

正解： B

解説：
Comprehensive and Detailed In-Depth Explanation:
As an Appian Lead Developer, preventing recurring "oddities and breaks" in application components requires addressing root causes-likely tied to human error, lack of oversight, or uncontrolled changes-while leveraging Appian's governance and collaboration features. The question implies a past mistake (e.g., accidental deletions or modifications) and seeks a proactive, sustainable solution. Let's evaluate each option based on Appian's official documentation and best practices:
A . Design and communicate a best practice that dictates designers only work within the confines of their own application:
This suggests restricting designers to their assigned applications via a policy. While Appian supports application-level security (e.g., Designer role scoped to specific applications), this approach relies on voluntary compliance rather than enforcement. It doesn't directly address "oddities and breaks"-e.g., a designer could still mistakenly alter components within their own application. Appian's documentation emphasizes technical controls and process rigor over broad guidelines, making this insufficient as a guarantee.
B . Ensure that the application administrator group only has designers from that application's team:
This involves configuring security so only team-specific designers have Administrator rights to the application (via Appian's Security settings). While this limits external interference, it doesn't prevent internal mistakes (e.g., a team designer deleting a critical component). Appian's security model already restricts access by default, and the issue isn't about unauthorized access but rather component integrity. This step is a hygiene factor, not a direct solution to the problem, and fails to "guarantee" prevention.
C . Create a best practice that enforces a peer review of the deletion of any components within the application:
This is the best choice. A peer review process for deletions (e.g., process models, interfaces, or records) introduces a checkpoint to catch errors before they impact the application. In Appian, deletions are permanent and can cascade (e.g., breaking dependencies), aligning with the "oddities and breaks" described. While Appian doesn't natively enforce peer reviews, this can be implemented via team workflows-e.g., using Appian's collaboration tools (like Comments or Tasks) or integrating with version control practices during deployment. Appian Lead Developer training emphasizes change management and peer validation to maintain application stability, making this a robust, preventive measure that directly addresses the root cause.
D . Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and

assign them the permissions to administer their application:
This option is confusingly worded but seems to suggest granting Designer system role permissions (a high-level privilege) while limiting developers to Viewer rights system-wide, with Administrator rights only for their application. In Appian, the "Designer" system role grants broad platform access (e.g., creating applications), which contradicts "basic user rights" (Viewer role). Regardless, adjusting permissions doesn't prevent mistakes-it only controls who can make them. The issue isn't about access but about error prevention, so this option misses the mark and is impractical due to its contradictory setup.
Conclusion: Creating a best practice that enforces a peer review of the deletion of any components (C) is the strongest solution. It directly mitigates the risk of "oddities and breaks" by adding oversight to destructive actions, leveraging team collaboration, and aligning with Appian's recommended governance practices. Implementation could involve documenting the process, training the team, and using Appian's monitoring tools (e.g., Application Properties history) to track changes-ensuring mistakes are caught before deployment. This provides the closest guarantee to preventing recurrence.
Reference:
Appian Documentation: "Application Security and Governance" (Change Management Best Practices).
Appian Lead Developer Certification: Application Design Module (Preventing Errors through Process).
Appian Best Practices: "Team Collaboration in Appian Development" (Peer Review Recommendations).

質問 # 40
You are planning a strategy around data volume testing for an Appian application that queries and writes to a MySQL database. You have administrator access to the Appian application and to the database. What are two key considerations when designing a data volume testing strategy?

- A. The amount of data that needs to be populated should be determined by the project sponsor and the stakeholders based on their estimation.
- B. Data model changes must wait until towards the end of the project.
- C. Testing with the correct amount of data should be in the definition of done as part of each sprint.
- D. Data from previous tests needs to remain in the testing environment prior to loading prepopulated data.
- E. Large datasets must be loaded via Appian processes.

正解：A、C

解説：
Comprehensive and Detailed In-Depth Explanation:Data volume testing ensures an Appian application performs efficiently under realistic data loads, especially when interacting with external databases like MySQL. As an Appian Lead Developer with administrative access, the focus is on scalability, performance, and iterative validation. The two key considerations are:
* Option C (The amount of data that needs to be populated should be determined by the project sponsor and the stakeholders based on their estimation):Determining the appropriate data volume is critical to simulate real-world usage. Appian's Performance Testing Best Practices recommend collaborating with stakeholders (e.g., project sponsors, business analysts) to define expected data sizes based on production scenarios. This ensures the test reflects actual requirements-like peak transaction volumes or record counts-rather than arbitrary guesses. For example, if the application will handle 1 million records in production, stakeholders must specify this to guide test data preparation.
* Option D (Testing with the correct amount of data should be in the definition of done as part of each sprint):Appian's Agile Development Guide emphasizes incorporating performance testing (including data volume) into the Definition of Done (DoD) for each sprint. This ensures that features are validated under realistic conditions iteratively, preventing late-stage performance issues. With admin access, you can query/write to MySQL and assess query performance or write latency with the specified data volume, aligning with Appian's recommendation to "test early and often."
* Option A (Data from previous tests needs to remain in the testing environment prior to loading prepopulated data):This is impractical and risky. Retaining old test data can skew results, introduce inconsistencies, or violate data integrity (e.g., duplicate keys in MySQL). Best practices advocate for a clean, controlled environment with fresh, prepopulated data per test cycle.
* Option B (Large datasets must be loaded via Appian processes):While Appian processes can load data, this is not a requirement. With database admin access, you can use SQL scripts ortools like MySQL Workbench for faster, more efficient data population, bypassing Appian process overhead.
Appian documentation notes this as a preferred method for large datasets.
* Option E (Data model changes must wait until towards the end of the project):Delaying data model changes contradicts Agile principles and Appian's iterative design approach. Changes should occur as needed throughout development to adapt to testing insights, not be deferred.
References:Appian Lead Developer Training - Performance Testing Best Practices, Appian Documentation - Data Management and Testing Strategies.

質問 # 41
Your application contains a process model that is scheduled to run daily at a certain time, which kicks off a user input task to a specified user on the 1st time zone for morning data collection. The time zone is set to the (default) pm!timezone. In this situation, what does the pm!timezone reflect?

- A. The time zone of the server where Appian is installed.
- B. The time zone of the user who is completing the input task.
- C. The time zone of the user who most recently published the process model.
- D. The default time zone for the environment as specified in the Administration Console.

正解： D

解説：
Comprehensive and Detailed In-Depth Explanation:
In Appian, the pm!timezone variable is a process variable automatically available in process models, reflecting the time zone context for scheduled or time-based operations. Understanding its behavior is critical for scheduling tasks accurately, especially in scenarios like this where a process runs daily and assigns a user input task.
Option C (The default time zone for the environment as specified in the Administration Console):
This is the correct answer. Per Appian's Process Model documentation, when a process model uses pm!timezone and no custom time zone is explicitly set, it defaults to the environment's time zone configured in the Administration Console (under System > Time Zone settings). For scheduled processes, such as one running "daily at a certain time," Appian uses this default time zone to determine when the process triggers. In this case, the task assignment occurs based on the schedule, and pm!timezone reflects the environment's setting, not the user's location.
Option A (The time zone of the server where Appian is installed): This is incorrect. While the server's time zone might influence underlying system operations, Appian abstracts this through the Administration Console's time zone setting. The pm!timezone variable aligns with the configured environment time zone, not the raw server setting.
Option B (The time zone of the user who most recently published the process model): This is irrelevant. Publishing a process model does not tie pm!timezone to the publisher's time zone. Appian's scheduling is system-driven, not user-driven in this context.
Option D (The time zone of the user who is completing the input task): This is also incorrect. While Appian can adjust task display times in the user interface to the assigned user's time zone (based on their profile settings), the pm!timezone in the process model reflects the environment's default time zone for scheduling purposes, not the assignee's.
For example, if the Administration Console is set to EST (Eastern Standard Time), the process will trigger daily at the specified time in EST, regardless of the assigned user's location. The "1st time zone" phrasing in the question appears to be a typo or miscommunication, but it doesn't change the fact that pm!timezone defaults to the environment setting.

質問 # 42
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: DEV2 > TEST (SIT/UAT) > PROD
- B. Phase 2 development work stream: DEV > TEST (SIT/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: DEV > TEST2 (SIT/UAT) > PROD
- D. Phase 2 development work stream: DEV > TEST (SIT) > STAGE (UAT) > PROD Production support work stream: DEV2 > STAGE (SIT/UAT) > PROD

正解： B

解説：
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.

質問 # 43
......

www.xhs1991.com◂サイトにて最新➡ ACD301 □□□問題集をダウンロードACD301合格体験談

- 有難いACD301｜権威のあるACD301資格トレーリング試験｜試験の準備方法Appian Lead Developer復習テキスト □ ウェブサイト「 www.goshiken.com 」を開き、“ ACD301 ”を検索して無料でダウンロードしてくださいACD301テキスト
- ACD301復習資料 □ ACD301認定内容 □ ACD301資格難易度 □ 今すぐ⇒ www.passtest.jp ⇐で[ACD301]を検索して、無料でダウンロードしてくださいACD301受験記
- ncon.edu.sa, www.stes.tyc.edu.tw, hometechlk.com, lms.ait.edu.za, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, myportal.utt.edu.tt, www.stes.tyc.edu.tw, www.stes.tyc.edu.tw, cocoasr18.blogspot.com, learningmart.site, gr-ecourse.eurospeak.eu, Disposable vapes

BONUS！！！ Tech4Exam ACD301ダンプの一部を無料でダウンロード：https://drive.google.com/open?id=1NDwiSTb15zI_GuEOtnLX7xM37CtWmz-L