Vendor: Salesforce

Exam Code: Plat-Dev-301

Exam Name: Salesforce Certified Platform Developer II - Multiple Choice

Version: DEMO

QUESTION 1
A developer needs to send Account records to an external system for backup purposes. The process must take a snapshot of Accounts as they are saved and then make a callout to a RESTful web service. The web service can only receive, at most, one record per call. What should a developer do to implement these requirements?

A. Implement platform events.
B. Create a future method.
C. Implement the Queueable interface.
D. Expose an Apex class as a web service.

Answer: C
Explanation:
This requirement involves two specific constraints: performing a callout after a database change (DML) and handling high-volume processing where the external service only accepts one record at a time. In Salesforce, you cannot perform a callout in the same transaction after a DML operation; you must use asynchronous processing.

The Queueable interface is the superior choice here over @future methods. Queueable Apex allows for "chaining" jobs. Since the external web service only accepts one record per call, a developer can write a Queueable class that processes the first record, makes the callout, and then checks if there are remaining records. If more records exist, the class can enqueue a new instance of itself to process the next record. This creates a controlled, sequential execution path that respects the external system's limits.

While @future methods can perform callouts, they cannot be chained and have no built-in way to handle a sequential "one-by-one" requirement for an unknown number of records efficiently. Platform events (Option A) are used for event-driven architectures but would still require an asynchronous consumer to handle the callout limits. Exposing a web service (Option D) is for receiving data, not sending it. Therefore, the Queueable interface provides the necessary flexibility and transaction control to manage the snapshot and individual callout requirements effectively.

QUESTION 2
Universal Containers uses a custom Lightning page to provide a mechanism to perform a step-by-step wizard search for Accounts. One of the steps in the wizard is to allow the user to input text into a text field, ERP_Number__c, that is then used in a query to find matching Accounts.

```
erpNumber = erpNumber + '%';

List<Account> accounts = [SELECT Id, Name FROM Account WHERE
ERP_Number__c LIKE :erpNumber];
```

A developer receives the exception 'SOQL query not selective enough'. Which step should be taken to resolve the issue?

A. Move the SOQL query to within an asynchronous process.
B. Mark the ERP_Number__c field as required.
C. Mark the ERP_Number__c field as an external ID.
D. Change the query to use a SOSL statement instead of SOQL.

Answer: C
Explanation:
The "SOQL query not selective enough" error occurs when a query on an object with a large

volume of records (typically over 200,000) does not use a filtered index to narrow down the result set effectively. In Salesforce, the query optimizer must be able to use an index to scan a small enough percentage of the total records to maintain performance. When a query filters on a non-indexed field, the system is forced to perform a full table scan, which triggers this exception to prevent performance degradation.

To resolve this, the developer must ensure that the field used in the WHERE clause is indexed. Marking a custom field as an "External ID" or "Unique" automatically creates a custom index on that field. By marking ERP_Number__c as an External ID, the platform generates a underlying database index, allowing the query optimizer to quickly locate matching records without scanning the entire table. While the LIKE operator with a trailing wildcard (e.g., 'Text%') can utilize an index, it will only be considered "selective" if the filter narrow the results below the platform's selectivity thresholds (typically 10% of the first million records). Making the field an External ID is the standard programmatic solution to provide the necessary indexing for selectivity.

QUESTION 3
Universal Containers is leading a development team that follows the source-driven development approach in Salesforce. As part of their continuous integration and delivery (CI/CD) process, they need to automatically deploy changes to multiple environments, including sandbox and production. Which mechanism or tool would best support their CI/CD pipeline in source-driven development?

A. Salesforce CLI with Salesforce DX
B. Change Sets
C. Salesforce Extensions for Visual Studio Code
D. Ant Migration Tool

Answer: A
Explanation:
Source-driven development shifts the "source of truth" from the Salesforce org to a version control system (like Git). Salesforce DX (Developer Experience) was specifically designed to support this paradigm by introducing the Salesforce CLI (Command Line Interface). The Salesforce CLI is the primary engine for modern CI/CD pipelines because it allows for the scripted creation of scratch orgs, automated testing, and the deployment of source code to any environment (sandboxes, scratch orgs, or production) using simple command-line instructions.

Unlike traditional tools, the Salesforce CLI supports the "Source" format, which is more granular and easier to manage in version control than the traditional Metadata format used by the Ant Migration Tool.

QUESTION 4
A developer is writing code that requires making callouts to an external web service. Which scenario necessitates that the callout be made in an asynchronous method?

A. The callout could take longer than 60 seconds to complete.
B. The callouts will be made in an Apex trigger.
C. Over 10 callouts will be made in a single transaction.
D. The callouts will be made using the REST API.

Answer: B
Explanation:
In Salesforce, a critical restriction is that synchronous callouts cannot be made from within an Apex trigger. This architectural limitation is in place because triggers execute as part of a

database transaction. Allowing a synchronous callout--which waits for a response from an external system--would keep the database connection and transaction locks open for an indeterminate amount of time. This could lead to severe performance bottlenecks and row-locking issues across the entire organization. Therefore, if a developer needs to initiate an external integration based on a record change (trigger), the logic must be handed off to an asynchronous process, such as a @future(callout=true) method, a Queueable class, or a Platform Event.

QUESTION 5
A developer is tasked with ensuring that email addresses entered into the system for Contacts and for a custom object called Survey_Response__c do not belong to a list of blocked domains. The list of blocked domains is stored in a custom object for ease of maintenance by users. The Survey_Response__c object is populated via a custom Visualforce page. What is the optimal way to implement this?

A. Implement the logic in validation rules on the Contact and the Survey_Response__c objects.
B. Implement the logic in a helper class that is called by an Apex trigger on Contact and from the custom Visualforce page controller.
C. Implement the logic in an Apex trigger on Contact and also implement the logic within the custom Visualforce page controller.

Answer: B
Explanation:
The requirement involves enforcing a business rule across two different objects (Contact and Survey_Response__c) and through different entry points: standard UI/Triggers for Contacts and a custom Visualforce page for Survey Responses. Implementing the logic separately in a trigger and a controller (Option C) is a poor architectural choice because it duplicates code, leading to increased maintenance effort and a higher risk of inconsistency if the logic needs to change in the future. Validation rules (Option A) are also unsuitable here because they cannot natively perform a dynamic lookup against a list of records in a separate "Blocked Domains" object without complex and unscalable workarounds.

The optimal approach is to centralize the validation logic within a single "Helper" or "Service" class. This follows the DRY (Don't Repeat Yourself) principle of software development. By creating a shared method in a helper class, the developer can ensure that both the Contact trigger and the Visualforce controller use the exact same logic to verify email domains. This centralized method queries the custom object containing the blocked domains and returns a validation result. This architecture simplifies testing, ensures uniform enforcement of business rules across the entire platform, and allows administrators to update the blocked domains list without requiring any further code modifications.

QUESTION 6
Universal Charities (UC) uses Salesforce to collect electronic donations in the form of credit card deductions from individuals and corporations. When a customer service agent enters the credit card information, it must be sent to a 3rd-party payment processor for the donation to be processed. UC uses one payment processor for individuals and a different one for corporations. What should a developer use to store the payment processor settings for the different payment processors, so that their system administrator can modify the settings once they are deployed, if needed?

A. Hierarchy custom setting
B. Custom label
C. Custom metadata
D. List custom setting

Answer: C
Explanation:
For storing application configurations and integration settings that need to be easily modified by administrators and deployed across environments, Custom Metadata Types are the preferred solution. Unlike Custom Settings (Options A and D), records within a Custom Metadata Type are considered metadata rather than data. This is a critical distinction for the development lifecycle because these records can be included in Change Sets or deployment packages. This eliminates the manual overhead and risk associated with re-creating configuration records in production after a sandbox deployment.

In this scenario, UC needs to manage settings for two different payment processors. A Custom Metadata Type can be created with fields for API endpoints, merchant IDs, and security keys. An administrator can then create and edit the specific records for the "Individual" and "Corporate" processors directly in the Setup menu. Furthermore, Custom Metadata queries are efficient and do not count against standard SOQL governor limits in many contexts. While Custom Labels (Option B) are useful for translating text, they are not intended for complex, structured configuration data. Hierarchy Custom Settings are designed for user-specific overrides, which is not applicable here. Therefore, Custom Metadata provides the most robust, deployable, and administrator-friendly way to manage external service configurations.

QUESTION 7
What should be added to the setup, in the location indicated, for the unit test above to create the controller extension for the test?

```
@isTest

static void testUpdateSuccess() {

Account acet = new Account(Name = 'test');

insert acet;

// Add code here

extension.inputValue = 'test';

PageReference pageRef = extension.update();

System.assertNotEquals(null, pageRef);

}
```

A. AccountControllerExt extension = new AccountControllerExt(acet);
B. ApexPages.StandardController sc = new ApexPages.StandardController(acet);
AccountControllerExt extension = new AccountControllerExt(sc);
C. ApexPages.StandardController sc = new ApexPages.StandardController(acet.Id);
AccountControllerExt extension = new AccountControllerExt(sc);
D. AccountControllerExt extension = new AccountControllerExt(acet.Id);

Answer: B
Explanation:
To properly test a Visualforce Controller Extension in Apex, the developer must replicate the instantiation process used by the Lightning Platform at runtime. A controller extension is a class

whose constructor is specifically defined to accept a single parameter of the type ApexPages.StandardController. The standard controller acts as the bridge between the custom extension logic and the underlying Salesforce record (in this case, an Account).

In the provided test method, a record acet has been inserted into the database. To create the extension object, the developer must first instantiate the StandardController by passing the acet SObject record into its constructor. Once the standard controller instance (e.g., sc) is created, it is then passed into the constructor of the AccountControllerExt class. Option B is the correct implementation of this two-step pattern. Option C is incorrect because the StandardController constructor requires a full SObject record, not just an ID string. Options A and D are incorrect because extension classes do not support constructors that take an SObject or an ID directly; they specifically require the StandardController object to provide the necessary context for standard actions and fields. Following the correct pattern ensures the test accurately simulates the page execution context.

QUESTION 8
How should a developer assert that a trigger with an asynchronous process has successfully run?

A. Create all test data, use @future in the test class, then perform assertions.
B. Create all test data in the test class, use System.runAs() to invoke the trigger, then perform assertions.
C. Create all test data in the test class, invoke Test.startTest() and Test.stopTest() and then perform assertions.
D. Insert records into Salesforce, use seeAllData=true, then perform assertions.

Answer: C
Explanation:
Testing asynchronous Apex (such as @future methods, Batchable, or Queueable classes) requires a specific mechanism to ensure that background tasks complete before the test verifies the results. In a standard execution, asynchronous jobs are queued and run whenever system resources are available, which means a test's assertion statements might execute before the background job has even started. To address this, Salesforce provides the Test.startTest() and Test.stopTest() methods.

When the code that initiates an asynchronous process is placed between Test.startTest() and Test.stopTest(), the platform behaves differently. All asynchronous calls made within this block are collected and held by the system. As soon as the code reaches Test.stopTest(), the test execution pauses, and the system forces all queued asynchronous jobs to run immediately and synchronously within the same thread. Once Test.stopTest() finishes, the execution continues to the next line. By placing assertions immediately after Test.stopTest(), the developer is guaranteed that the asynchronous logic has finished its work. This is the only supported way to reliably test the side effects of background processes. Other options, like using System.runAs() or seeAllData=true, do not affect the timing of asynchronous execution and will likely result in failed assertions.

QUESTION 9
A developer wrote a class named AccountHistoryManager that relies on field history tracking. The class has a static method called getAccountHistory that takes in an Account as a parameter and returns a list of associated AccountHistory object records. The following test fails:

```
@isTest

public static void testAccountHistory(){
```

```
Account a = new Account(name = 'test');

insert a;

a.name = a.name + '1';

update a;

List<AccountHistory> ahList =
AccountHistoryManager.getAccountHistory(a);

System.assert(ahList.size() > 0);

}
```

What should be done to make this test pass?

A. Use @isTest(SeeAllData=true) to see historical data from the org and query for AccountHistory records.
B. Use Test.isRunningTest() in getAccountHistory() to conditionally return fake AccountHistory records.
C. The test method should be deleted since this code cannot be tested.
D. Create AccountHistory records manually in the test setup and write a query to get them.

Answer: B
Explanation:
A significant challenge in Salesforce unit testing is that system-generated records, such as those in the AccountHistory or OpportunityHistory objects, are not created during the execution of a test method, even if field history tracking is enabled in the organization. Because these records are read-only and managed by the system, a developer cannot manually insert them via DML within a test setup. Consequently, the query in the getAccountHistory method will always return an empty list in a test context, causing the System.assert to fail.

The optimal programmatic solution to this limitation is to implement a "mocking" strategy using Test.isRunningTest(). By modifying the production code within getAccountHistory, the developer can check if the code is currently being executed by a unit test. If it is, the method can return a manually constructed list of AccountHistory records (stored in memory but not inserted into the database) to satisfy the test's requirements. This allows the test to verify the logic that processes the history data without needing the system to actually generate historical records. While SeeAllData=true (Option A) would allow the test to see real data in the org, it is considered a poor practice because it makes the test dependent on the specific state of the organization's data, leading to brittle tests that may fail in different environments.

QUESTION 10
Consider the following code snippet:

```
trigger OpportunityTrigger on Opportunity (before insert, before
update) {

for(Opportunity opp : Trigger.new){

OpportunityHandler.setPricingStructure(Opp);

}
```

```
}

public class OpportunityHandler{

public static void setPricingStructure(Opportunity thisOpp){

Pricing_Structure_c ps = [Select Type_c FROM Pricing_Structure_c WHERE
industry_c = :thisOpp.Account_Industry_c];

thisOpp.Pricing_Structure_c = ps.Type_c;

update thisOpp;

}

}
```

Which two best practices should the developer implement to optimize this code?

A. Remove the DML statement.
B. Change the trigger context to after update, after insert.
C. Query the Pricing_Structure_c records outside of the loop.
D. Use a collection for the DML statement.

Answer: AC
Explanation:
The provided code violates two fundamental Salesforce Apex best practices: bulkification and efficient use of trigger contexts. First, the code performs a SOQL query inside a loop (setPricingStructure is called inside the trigger's for loop). This is a critical performance issue because it consumes one SOQL query per record in the transaction. If 200 opportunities are updated at once, the code will hit the governor limit of 100 SOQL queries and throw an exception. To optimize this, the developer should use a Map or List to query all necessary Pricing_Structure__c records once, outside the loop (Option C), and then reference that collection during processing.

Second, the code is running in a before trigger context. In a before insert or before update trigger, any changes made directly to the records in Trigger.new are automatically saved to the database when the trigger completes without the need for an explicit DML update statement. Including an update thisOpp; statement (Option A) is not only redundant but dangerous, as it can trigger a recursive loop where the update re-fires the same trigger, leading to a "Maximum stack depth reached" error. By removing the DML statement and bulkifying the query, the code becomes more efficient, scalable, and compliant with platform governor limits.

QUESTION 11
A developer is tasked with creating a Lightning web component that allows users to create a Case for a selected product, directly from a custom Lightning page. The input fields in the component are displayed in a non-linear fashion on top of an image of the product to help the user better understand the meaning of the fields. Which two components should a developer use to implement the creation of the Case from the Lightning web component?

A. lightning-record-form
B. lightning-record-edit-form
C. lightning-input-field

D. lightning-input

Answer: BC
Explanation:
When a requirement specifies a "non-linear" or highly customized layout, the high-level lightning-record-form (A) is unsuitable because it automatically renders fields in a standard grid or based on a page layout. To achieve a custom UI--such as overlaying fields on a product image--the developer needs more granular control.

The lightning-record-edit-form (B) is the ideal container for this scenario. It provides the heavy lifting for data communication (loading and saving records) without imposing a strict visual structure. Within this form, the developer uses lightning-input-field (C) components for each Case field. These components are "context-aware," meaning they automatically inherit metadata like field labels, picklist values, and validation rules directly from the Salesforce object definition.

Because lightning-input-field components can be placed anywhere inside the lightning-record-edit- form tag, the developer can use custom CSS (absolute positioning or CSS Grid) to place them precisely over the product image as requested. Standard lightning-input (D) components could work but would require much more manual code to handle data binding, type conversion, and validation, making the combination of B and C the most efficient and platform-aligned approach.

QUESTION 12
A large company uses Salesforce across several departments. Each department has its own Salesforce Administrator. It was agreed that each Administrator would have their own sandbox in which to test changes. Recently, users notice that fields that were recently added for one department suddenly disappear without warning. Which two statements are true regarding these issues and resolution?

A. The administrators are deploying their own Change Sets, thus deleting each other's fields from the objects in production.
B. The administrators are deploying their own Change Sets over each other, thus replacing entire Page Layouts in production.
C. Page Layouts should never be deployed via Change Sets, as this causes Field-Level Security to be reset and fields to disappear.
D. A sandbox should be created to use as a unified testing environment instead of deploying Change Sets directly to production.

Answer: BD
Explanation:
This scenario highlights a common conflict in multi-admin environments known as "the last delivery wins" problem. When multiple administrators work in isolated sandboxes, they are essentially working on different versions of the same metadata. If Admin A adds a field to a Page Layout and deploys it, and then Admin B--who does not have Admin A's changes in their sandbox--deploys their own version of that same Page Layout, Admin B's deployment will overwrite Admin A's changes in Production (Statement B). In the Metadata API (which Change Sets use), Page Layouts are treated as single files; you cannot "merge" them via a Change Set; you can only replace the destination file entirely.

To resolve this, the team needs a more sophisticated deployment pipeline. Statement D is the correct resolution: the company should implement a "Staging" or "Integration" sandbox. In this model, all admins deploy their Change Sets to a single unified sandbox first. This allows them to identify conflicts and "merge" their changes (manually or via source control) before a final, combined deployment is made to Production. Statement A is technically incorrect because

Change Sets cannot "delete" components; they can only overwrite or add. Statement C is a myth; Page Layouts can be safely deployed, but only if the underlying fields and security settings are also included and coordinated.

QUESTION 13
A company has a Lightning page with many Lightning Components, some that cache reference data. It is reported that the page does not always show the most current reference data. What can a developer use to analyze and diagnose the problem in the Lightning page?

A. Salesforce Lightning Inspector Storage tab
B. Salesforce Lightning Inspector Event Log tab
C. Salesforce Lightning Inspector Actions tab
D. Salesforce Lightning Inspector Transactions tab

Answer: A
Explanation:
The Salesforce Lightning Inspector is a Chrome DevTools extension used to investigate and optimize Lightning components. When a page fails to show the most current reference data, the issue often lies within the client-side caching mechanism. The Storage tab (A) is specifically designed to help developers inspect the client-side data stores used by the Lightning Data Service (LDS) and custom component caching.

In this tab, a developer can view the contents of the various storage caches (such as actions, records, and lists). It allows the developer to see exactly what data is currently stored in the cache, its expiration time, and whether the data is being retrieved from the server or served directly from the cache. By analyzing the Storage tab, a developer can determine if a component is incorrectly holding onto "stale" data instead of refreshing from the server.

The Actions tab (C) is useful for monitoring the status of server-side controller actions, and the Event Log (B) tracks the sequence of application and component events. However, for a data consistency issue specifically related to caching, the Storage tab provides the most direct visibility into the underlying cause. Using this tool, a developer can verify if the cache invalidation logic is working correctly or if the storage limits are causing unexpected data eviction.

QUESTION 14
Consider the following code snippet:

```
<apex:page docType="html-5.0" controller="FindOpportunities">

<apex:form >

<apex:pageBlock >

<apex:pageBlockSection title="find opportunity">

<apex:input label="opportunity name"/>

<apex:commandButton value="search" action="{!search}"/>

</apex:pageBlockSection>

<apex:pageBlockSection title="Opportunity List" id="opportunityList">
```

```
</apex:pageBlockSection>

</apex:pageBlock>

</apex:form>

</apex:page>
```

Users of this Visualforce page complain that the page does a full refresh every time the Search button is pressed. What should the developer do to ensure that a partial refresh is made so that only the section identified with opportunityList is re-drawn on the screen?

A. Enclose the DATA table within the <apex:actionRegion> tag.
B. Implement the <apex:actionFunction> tag with immediate = true.
C. Ensure the action method search returns null.
D. Implement the reRender attribute on the <apex:commandButton> tag.

Answer: D
Explanation:
In Visualforce, the standard behavior for an <apex:commandButton> is to perform a full page postback. To convert this into an asynchronous (AJAX) request that only updates a specific portion of the page, the developer must use the reRender attribute (D). By setting reRender="opportunityList" on the command button, the developer instructs the platform to capture the server's response and update only the DOM element with that specific ID, rather than refreshing the entire browser window.

QUESTION 15
Given a list of Opportunity records named opportunityList, which code snippet is best for querying all Contacts of the Opportunity's Account?

A. List <Contact> contactList = new List <Contact>(); for(Opportunity o : opportunityList){ Account a = [SELECT Id, (SELECT Id FROM Contacts) FROM Account WHERE Id = :o.AccountId] contactList.addAll(a.Contacts);)
B. List <Contact> contactList = new List <Contact>(); Set <Id> accountIds = new Set <Id> (); for (Opportunity o : opportunityList){ contactIds.add(o.ContactId); } for(Contact c : [SELECT Id FROM Contact WHERE Id IN :contactIds]){ contactList.add(c); }
C. List <Contact> contactList = new List <Contact>(); Set <Id> accountIds = new Set <Id> (); for(Opportunity o : opportunityList){ accountIds.add(o.AccountId); } for(Account a : [SELECT Id, (SELECT Id FROM Contacts) FROM Account WHERE Id IN :accountIds]){ contactList.addAll(a.Contacts); }
D. List <Contact> contactList = new List <Contact>(); for (Contact c : [SELECT Id FROM Contact WHERE AccountId IN :opportunityList.AccountId]){ contactList.add(c); }

Answer: C

QUESTION 16
The Account after-update trigger fires whenever an Account's address is updated, and it updates every associated Contact with that address. The Contact after-update trigger fires on every edit, and it updates every Campaign Member record related to the Contact with the Contact's state. Consider the following: A mass update of 200 Account records' addresses, where each Account has 50 Contacts. Each Contact has one Campaign Member. This means there are 10,000 Contact records across the Accounts and 10,000 Campaign Member records across the contacts. What will happen when the mass update occurs?

A. The mass update of Account address will succeed, but the Contact address updates will fail due to exceeding number of records processed by DML statements.
B. There will be no error, since each trigger fires within its own context and each trigger does not exceed the limit of the number of records processed by DML statements.
C. The mass update will fail, since the two triggers fire in the same context, thus exceeding the number of records.

Answer: C
Explanation:
In Salesforce, all triggers that fire as a result of an initial DML operation run within the same execution context and the same transaction. This means that all governor limits--such as the limit of 10,000 records processed by DML statements--are shared across the entire chain of events.

In this scenario, the transaction begins with the update of 200 Accounts. The Account trigger then attempts to update 10,000 related Contacts (50 Contacts per Account across 200 Accounts). This update of 10,000 Contacts then fires the Contact trigger, which attempts to update 10,000 related Campaign Members. When the system attempts to process these updates, the total number of records processed via DML in a single transaction will exceed the platform limit of 10,000 rows.

Even though the Account update is only 200 records, the subsequent cascading updates (10,000 Contacts + 10,000 Campaign Members) bring the total DML count to 20,200. This triggers a LimitException, and because Apex transactions are atomic, the entire operation will fail and roll back. To resolve this, the developer would need to handle these updates asynchronously (e.g., using Batch Apex or Queueable) to split the cascading updates into separate transactions with their own sets of governor limits.

QUESTION 17
Which statement is considered a best practice for writing bulk safe Apex triggers?

A. Add LIMIT 50000 to every SOQL statement.
B. Add records to collections and perform DML operations against these collections.
C. Use the Database methods with allOrNone set to false instead of DML statements.
D. Perform all DML operations from within a future method.

Answer: B
Explanation:
The most fundamental best practice for writing bulk-safe (scalable) Apex triggers is to avoid performing DML operations or SOQL queries inside of loops. Salesforce enforces a limit of 150 DML statements per transaction. If a trigger iterates through a list of 200 records and performs a delete or update inside that loop, the code will fail with a governor limit exception after the 150th record.

To make a trigger bulk-safe, developers must collect all records that need to be processed into a collection (such as a List or Map) and then perform a single DML operation on that entire collection outside of the loop (Option B). This consumes only one DML statement regardless of the number of records (up to the 10,000 row limit).

Thank You for Trying Our Product

Lead2pass Certification Exam Features:

★ More than **99,900** Satisfied Customers Worldwide.

★ Average **99.9%** Success Rate.

★ **Free Update** to match latest and real exam scenarios.

★ **Instant Download** Access! No Setup required.

★ Questions & Answers are downloadable in **PDF** format and **VCE** test engine format.

★ Multi-Platform capabilities - **Windows, Laptop, Mac, Android, iPhone, iPod, iPad**.

★ **100%** Guaranteed Success or **100%** Money Back Guarantee.

★ **Fast**, helpful support **24x7**.

View list of all certification exams: http://www.lead2pass.com/all-products.html

10% Discount Coupon Code: **ASTR14**