Automating Apex Test Data Generation Using AI Models

Published August 19, 2025 50 min read

AI-Driven Test Data Synthesizer for Apex Unit Tests

Introduction

Salesforce developers often spend significant effort writing Apex unit tests to satisfy the 75% code coverage requirement for deployments (Source: developer.salesforce.com). A common pain point is setting up **test data** – creating realistic records like Accounts, Contacts, and custom objects – to exercise business logic in isolation (Source: blog.beyondthecloud.dev). Poor data design, many validation rules, and complex triggers can make test data setup cumbersome, often causing developers to get "stuck" writing lengthy @testSetup methods (Source: blog.beyondthecloud.dev). To boost productivity and consistency, we propose an AI-driven test data synthesizer that automatically generates both test method stubs and the necessary mock data. This tool takes an Apex class signature (its methods, parameters, and relevant field definitions) as

input, and uses a large language model (LLM) to output a complete test class with realistic test records and assertions. The goal is to automate repetitive unit test boilerplate, allowing developers to focus on business logic instead of manual data construction.

This report presents a detailed design for such a system, including architectural diagrams, integration with Salesforce APIs, and examples of AI-generated test classes. We discuss trade-offs between using prompt-based LLM code generation versus deterministic templates (Source: diffblue.com). We also address Salesforce-specific testing constraints – like governor limits, test isolation, and how to safely create or **mock** database objects. Best practices for generating DML-safe test data (for standard objects like `Account`/`Contact` and custom objects) are highlighted. We explore how the system can leverage the Salesforce Metadata API, Tooling API, and Schema describe calls to gather necessary context (class metadata and field requirements) and how it can deploy the generated tests. Finally, we consider the option to fine-tune models on Apex code corpora and compare using a local model vs. a hosted AI service. The aim is to provide architects and developers a comprehensive blueprint for an AI-assisted Apex test generator that improves developer productivity while producing high-quality, maintainable tests.

Solution Architecture Overview

At a high level, the AI-driven test generator can be envisioned as an extension to the Salesforce developer workflow (for example, a VS Code plugin or CLI tool) that interacts with both the **Salesforce platform** and an **LLM service**. **Figure 1** illustrates the architecture and data flow:

Figure 1: High-level architecture of the AI-driven Apex test synthesizer. The tool retrieves the Apex class signature and relevant schema details, composes a prompt for an LLM (e.g. GPT), and obtains a generated test class. The developer can review and refine the output before saving it back to the org via Metadata API or to the codebase.

Starting from the left of Figure 1, the developer triggers test generation (for example, by a command or UI action). The tool then performs the following steps:

1. **Ingest Apex Class Signature** – The target Apex class's definition is retrieved. This includes the class name, its methods (names, parameters, return types), and any properties or fields declared. The signature can be obtained either from local source (if the developer is in an IDE) or via the Salesforce **Metadata API** if only the class name is known. The Salesforce Metadata API allows programmatic retrieval of Apex classes by name (Source: trailhead.salesforce.com)(Source: salesforcebuddy.com), and can fetch the class body or a symbol table. Optionally, the **Tooling API** can be used to get a symbol table (structured JSON of the class's methods and attributes) for more formal parsing. At minimum, the tool identifies the public methods and their signatures that need covering.

2. **Retrieve Schema/Field Definitions** – To generate valid test records, the tool gathers schema metadata for any SObjects referenced by the class. For example, if a method accepts an `Account` or queries `Contact`, the tool should know required fields and data types for those objects. This can be done via Salesforce's schema describe calls (e.g. the REST API `sobjects/Account/describe` or Tooling API queries) to get field definitions (which fields are required, their types, picklist values, etc.). In practice, a simple approach is to use the SFDX CLI or Metadata API to retrieve the object's metadata. In one real-world example, developers copy-pasted the Fields list from Salesforce Object Manager into the prompt to help ChatGPT generate code (Source: salesforcedevops.net). Our tool can automate this by retrieving field info: for standard objects like Account (where at least the "Name" field is required) and for custom objects (which have a Name field and any custom required fields). This **schema context** will help the AI suggest realistic field values that won't violate not-null or format constraints (for instance, ensuring `Contact.LastName` is set, as it's required).

3. **Compose Prompt for LLM** – The class signature and schema details are combined into a prompt for the LLM. The prompt likely includes: a brief instruction to **"Generate an Apex test class"** for the given class, a summary of each method to test (and perhaps a description of the class's purpose if available via comments or naming), and a list of relevant object fields (especially required fields or important business fields). The style can be inspired by interactive scripting tutorials – e.g., Keir Bowden's blog samples – where the code generation might be done step by step in a conversational manner. However, in an automated tool, the prompt is crafted programmatically. It might look like: *"Given the following Apex class signature, produce an @isTest class with test methods. Class X has methods A(param1 Type1, param2 Type2)... The Account object has fields: Name (required), Industry (picklist), etc..."*. Including field metadata can guide the LLM to create correct and meaningful record data (e.g., setting `Account.Name = 'Test Account'` because Name is required). This approach was validated by a Salesforce devops expert who fed custom field XML definitions into ChatGPT to improve test factory generation (Source: salesforcedevops.net).

4. **Invoke LLM for Test Generation** – The prompt is sent to the chosen LLM (such as OpenAI's GPT-4 or a similar model). The LLM, having been trained on vast amounts of code including possibly Apex-like syntax, will generate Apex code for a test class. The output should include:

    - The test class declaration (`@isTest` class naming convention like `MyClassTest`).
    - Test method stubs for each public method in the original class. Each test method will create necessary test data, call the target method, then perform `System.assert` checks.
    - Realistic test data creation: e.g., inserting an `Account` with required fields populated, or constructing custom object records with dummy values. The model might also include edge-case tests (like passing nulls or empty lists) if prompted for thoroughness.

- Proper structure using best practices: use of `Test.startTest()` and `Test.stopTest()` around the execution under test (to reset governor limits and simulate separate transaction), and test method modifiers (`@isTest` or `testMethod` as needed).
- No use of org data (ensure `SeeAllData=false` either implicitly by default or explicitly set, to adhere to test isolation best practice (Source: levelupsalesforce.com)).

It's worth noting that Salesforce itself has introduced an AI-assisted test generation (Agentforce for Developers) that uses LLMs for Apex, but it currently *does not generate test data alongside the test code*(Source: developer.salesforce.com). Our tool specifically addresses that gap by synthesizing the test records as well, not just empty test methods.

5. **Review and Refinement** – The raw output from the LLM is presented to the developer for review. Given that LLMs can occasionally produce incorrect or suboptimal code (e.g. referencing non-existent fields or methods due to hallucination (Source: diffblue.com)), this step is important. The developer can be shown the generated test in an interactive panel (for example, similar to how Agentforce lets you "Accept" or "Try Again" a generated test (Source: developer.salesforce.com)). If something looks off – say the AI misunderstood a method's intent or missed an assertion – the developer can provide feedback or adjust the prompt and regenerate. This iterative, interactive scripting approach is inspired by how one might refine code in an interactive session (as seen in some of Keir Bowden's scripting examples). The tool could allow the user to enter a follow-up instruction, like *"Add a test for the negative path where the input is null"*, and send the refined prompt to the LLM, enabling a conversational improvement loop. This **feedback loop** (dashed line in Figure 1) leverages the LLM's ability to follow incremental instructions to fine-tune the output within the same context.

6. **Deployment of Test Class** – Once the developer is satisfied, the final test class code is saved. In a local development scenario (SFDX project), the class file can be created in the `/force-app/main/default/classes` directory. The Salesforce CLI (sfdx) or Metadata API can then deploy this class to the org for execution. Alternatively, if the tool is running in a web context or in org (e.g., a browser-based IDE or a DevOps pipeline), it could call the Metadata API's `deploy()` to push the new Apex class. The Salesforce Metadata API supports deploying Apex classes from source, and the Tooling API even allows creating ApexClass records directly. Proper API integration ensures the generated test is persisted in the developer's org or repository.

Finally, the new test can be executed to verify it passes and indeed covers the target class. The entire process, from input class to runnable test, could take a matter of seconds in the ideal case – dramatically accelerating the testing phase of development.

Prompt-Based Generation vs. Deterministic Templates

One architectural choice is whether to rely on a flexible LLM prompt or to use a deterministic code generation template. Our AI-driven approach emphasizes the former, but it's important to understand the trade-offs:

- **Prompt-Based LLM Generation**: Using an LLM (like GPT-4) means the output test class is generated by a learned model based on patterns in its training data. This offers *flexibility* and *contextual richness*. The AI might incorporate best-practice patterns or handle a variety of class structures intelligently without explicit programming. For example, given a method that does a SOQL query, it might automatically create records so that the query returns results, and even assert on fields of those results. However, LLM outputs are **non-deterministic** – the same prompt may yield slightly different code on different runs (Source: diffblue.com). Subtle rewording of the prompt or changes in model version can alter the output. This can lead to inconsistent tests if not carefully managed. There's also a risk of *hallucinations*, where the AI might invent field names or method behaviors that don't exist, causing compile errors or logic errors in tests (Source: diffblue.com). Despite these risks, prompt-based generation can produce very human-like, readable tests that potentially cover edge cases (if prompted to do so) and follow naming conventions or patterns seen in community code.

- **Deterministic Template Generation**: A deterministic approach would use predefined code templates or programmatic rules to generate tests. For instance, one could write a script that parses an Apex class AST (abstract syntax tree) and outputs a test class skeleton: for each public method, create a test method with the same name plus "_Test", set up minimal required records, call the method, and assert that the method executed (maybe by checking some result or side effect if it can be inferred). This approach guarantees **repeatability** – the same input always produces the same output (Source: diffblue.com). Tools like Diffblue Cover (for Java) exemplify deterministic test generation, using static analysis to systematically create tests (Source: diffblue.com)(Source: diffblue.com). The benefit is reliability: if you run it today or in 6 months on the same class, you get identical results, making tests easier to trust and maintain in CI pipelines (Source: diffblue.com)(Source: diffblue.com). It also avoids wild mistakes because the generation logic is fully under our control (no "AI creativity" to introduce random code). On the downside, template-based tests might be simplistic or not handle complex logic well. They might only cover happy-path scenarios unless the template logic itself becomes very sophisticated (essentially needing to simulate logic execution to determine expected outcomes, which is hard without an AI). Also, purely template-based solutions may require continual updates to handle new language features or patterns.

Comparative Trade-offs: In practice, a hybrid approach can be effective. For example, the tool could use a deterministic framework for the outer structure (ensuring consistent class naming, test method naming, boilerplate annotations) but call the LLM to fill in the inner logic of test methods (the specific records to create and assertions to make). This ensures basic consistency (one test class per target class, etc.) while leveraging AI for the non-trivial parts (like guessing what assertions are meaningful). The choice may also depend on use-case: if the priority is **fast, consistent baseline coverage**, deterministic generation might suffice to get to 75% coverage on all classes. If the goal is **developer assistance and productivity**, generating richer tests via AI (which the developer can refine) might add more value.

A summary comparison is given in **Table 1**:

APPROACH	BEHAVIOR	PROS	CONS
Prompt-based LLM Generation (e.g. GPT-4 or Code LLM writes the test from instructions)	Non-deterministic, learned patterns influence output:contentReference\[oaicite:20\]{index=20}	- Flexible, context-aware (can incorporate best practices) - Little upfront rule-coding required (model does the work) - Potentially more human-like and comprehensive tests	- Output can vary between runs; not repeatable:contentReference\[oaicite:21\]{index=21} - Risk of errors or hallucinated code:contentReference\[oaicite:22\]{index=22} - Requires careful prompt tuning and result validation
Deterministic Template Generation (e.g. static code analysis + predefined test patterns)	Consistent logic yields same output for same input:contentReference\[oaicite:23\]{index=23}	- Predictable and stable results (good for CI):contentReference\[oaicite:24\]{index=24} - No surprises: easier to debug and trust tests - No external model needed (can run fully offline)	- Requires developing and maintaining generation logic - May produce simplistic tests that miss edge cases - Harder to adapt to complex code without AI reasoning

In our design, we lean on LLM-based generation to maximize automation of non-trivial test logic. But we mitigate its downsides by anchoring the generation with real metadata (to reduce hallucination) and by involving the developer in review (catching any issues). Over time, usage of the tool could also reveal common patterns which we can feed back into the prompt or a fine-tuned model, gradually making the AI's output more deterministic in practice.

Salesforce Unit Testing Constraints and Considerations

Generating Apex tests is not just a code generation problem – one must respect the **Salesforce platform's testing constraints** and best practices. Our AI tool must be aware of these constraints to produce valid and efficient tests. Key considerations include:

- **Isolation of Test Data**: Apex tests run in their own context and cannot see the org's standard data or uncommitted records from other tests. By default, tests have `SeeAllData=false` (meaning no access to org records like Accounts or Contacts that aren't created in the test itself). The tool must ensure all data needed for the test is created within the test method or a @TestSetup method. It should avoid using any real IDs or querying for existing data (unless explicitly intended for certain integration tests, which is rare and not recommended). Each test method gets a fresh set of governor limits and a rollback at the end,

so the AI can freely insert and modify data knowing it will not persist or affect other tests (Source: levelupsalesforce.com). Starting with API v28, even things like the VLOOKUP formula in validation rules are isolated to test-created data (Source: concret.io), so the test should always supply its own records for any data dependencies.

- **Governor Limits**: Salesforce enforces strict limits in Apex, and tests are no exception. Each test method (each execution context) can execute at most 100 SOQL queries and 150 DML statements, among other limits (Source: salesforceben.com)(Source: salesforceben.com). The test generator needs to produce code that stays well within these limits. This usually isn't a problem for simple tests, but if the AI naively creates dozens of records in separate insert statements, it could inch toward limits or slow down test execution. Best practice is to **bulk insert** multiple records in one DML operation when possible (e.g., inserting a list of Contacts rather than one-by-one) to minimize DML count. Similarly, if the target class needs a lot of setup data, the test should use a single query or efficient loops to set it up, and not e.g. query inside a loop (a known "governor limit sin" (Source: salesforceben.com)). We can also instruct the LLM to utilize `Test.startTest()/Test.stopTest()`. Besides giving a fresh set of limits for the portion inside start/stop, this is crucial when testing asynchronous behavior or simply to separate setup from execution time. Our prompts should encourage the AI to wrap the method invocation in `Test.startTest(); ... Test.stopTest();`. This way, any SOQL/DML in the tested code doesn't count against the setup operations, and any async code (future methods, queueables, batch jobs) will run at `Test.stopTest()`. In summary, the generated test should be **governor-aware**: efficient data creation and proper use of test context to avoid hitting limits.

- **Performance and Bulk Testing**: Salesforce runs all tests, for example during deployments or CI. If our generator creates extremely heavy tests (e.g., inserting thousands of records or doing nested loops), it could slow down the test suite. We must balance realism with performance. For example, if testing a method that processes a list of records, it's enough to create a handful of records (maybe 2-3) to simulate bulk behavior, rather than hundreds. The AI should default to small but meaningful data sets (unless asked for stress tests). Salesforce also imposes a limit on total Apex test execution time and CPU time per test. So a best practice is to avoid unnecessary waits or huge computations in tests. The tool should not, for instance, generate an infinite loop or extremely large data by accident – again, careful prompt design and maybe post-processing checks are needed.

- **Test Class Isolation and Order Independence**: Each test class in Salesforce should not rely on another test class's data or execution order. The AI generator will produce a self-contained test class. If multiple test methods in the same class share common setup data (like a set of Accounts that many methods use), the generator can utilize an `@testSetup` method. This special method runs once per class before the test methods and can insert common records. Using `@testSetup` is a best practice to avoid duplicating setup code and to slightly improve performance (setup runs once, not before every method) (Source: salesforce.stackexchange.com). Our tool can detect if multiple tests in the class would need the same records and factor that into a single setup method. For example, if testing a service class with many methods all operating on Account and Contact, creating one Account in testSetup and related

Contacts can serve all methods. However, the generator must also understand that any modifications to those records in one test method do **not** persist into the next method (each test method gets a fresh copy of the @testSetup data). This usually doesn't require special handling unless the tests themselves are interdependent (which they shouldn't be). We simply ensure each test either uses the fresh testSetup data or creates its own, and does not assume any ordering.

- **Mocking Callouts and Stubbing Behavior**: Some Apex classes perform HTTP callouts or rely on platform features like Send Email, which require special handling in tests. If the class under test does a callout (using `Http.send()`), the test must implement an `HttpCalloutMock` and set `Test.setMock(...)` in order to simulate a response. Our generator could recognize from the method signature or body (if provided) that a callout is made (e.g., the presence of `HttpResponse` or `@future(callout=true)` annotation) and automatically generate a nested class implementing `HttpCalloutMock` with a dummy response. This would be an advanced capability, but very useful for completeness. Similarly, if the class queries `Schema.describe` or uses a static method that we might want to fake, we could use dependency injection or the Apex Stub API. Salesforce's Stub API allows creation of mock instances for interfaces or virtual classes at runtime (Source: developer.salesforce.com). For instance, if the class uses a custom interface to fetch data (the "data access layer" pattern), our test could use `Test.createStub` to provide a fake implementation (Source: softserveinc.com)(Source: softserveinc.com). However, automating this requires deeper analysis of the class, likely beyond just the signature. For now, a simpler approach is: if callouts are detected, generate a placeholder mock class; if the class is doing something complex like a database operation that could be abstracted, note it in comments for the user.

- **DML Safety and Data Validity**: "DML-safe test data" means the records we create in tests should satisfy all requirements so that inserts/updates succeed and do not trigger unintended side-effects. The AI must be aware of required fields – e.g., all standard objects require a Name (or LastName for Contact, etc.). If the target org has extra validation rules or triggers (beyond the tool's knowledge), generated data might cause those to fail. We can't know all custom logic, but we can follow general best practices:

  - Always set **required fields**: for example, `Account.Name`, `Contact.LastName`, `Opportunity.Name`, etc. If an object has record types, and the code under test might expect a certain record type, the test should set it or default to a valid one. (We might retrieve default RecordType Id via tooling API if needed, but that adds complexity; likely we assume the default is fine or the user adjusts the output).

  - Provide **valid field values**: e.g., if a field is a picklist, use a legitimate value. If our schema metadata tells us the picklist options, we could randomly choose one or always pick the first. For standard picklists like Industry on Account, the AI likely knows common values ("Technology", "Finance" etc.), but for custom ones, we might include one in the prompt.

- Avoid data that violates unique constraints or lookup relationships. If creating child records (like Contacts linked to Account), ensure the foreign key (AccountId) is set to an Account inserted in the test.
- Consider using utility methods like `Test.loadData` (which loads test records from a static CSV resource) for bulk data. Our tool might not automatically create CSVs, but for a very large setup, it could suggest this approach or generate the minimal code to load a resource if one is available. Generally, though, for AI generation, we focus on inline creation for simplicity.
- Clean up not needed, as tests rollback, but still we don't want to insert extraneous data. The AI should create only what's necessary for clarity and coverage.

By adhering to these constraints, the generated tests will not only compile but also run successfully in Salesforce. The developer can have confidence that accepting the AI's suggestion won't break their CI pipeline with governor limit errors or failing assertions. Our prompt to the AI will include some of these guidelines (implicitly or explicitly). For instance, we might add a prompt note: *"Ensure all required fields are populated and use Test.startTest/stopTest appropriately. The test should not perform more than a few DML operations."* If the AI still generates something problematic (like too many inserts in a loop), the review step allows catching that and adjusting.

Best Practices for Test Data Generation in Apex

In generating test data, our tool follows established best practices so that the output is idiomatic and robust. Here we summarize those best practices and how the AI tool implements them:

- **Use of Test Data Factory/Builder Patterns**: In large projects, a common practice is to use a **Test Data Factory** – a set of static methods or classes dedicated to creating test records (often using the *Builder* pattern for flexibility) (Source: blog.beyondthecloud.dev)(Source: blog.beyondthecloud.dev). For example, a `TestDataFactory.createAccount()` method that returns a ready-to-insert Account with all required fields. Our AI could detect if the project already has such utility classes (perhaps via naming conventions or scanning the codebase). If so, it might call those instead of duplicating record construction. However, in a greenfield scenario, the AI will produce explicit record creation code. It will create sObjects via the constructor syntax (e.g., `new Account(Name='Acme Inc')`). This is straightforward and lets the test class be self-contained. For teams that prefer using a builder library (like the open-source `apex-domainbuilder` or the example TDF in Piotr Gajek's blog (Source: blog.beyondthecloud.dev)), the tool could be configured to output using those APIs. For instance, instead of `insert new Account(Name='X')`, it might do `Account acct = (Account)TestDataFactory.AccountBuilder().withName('X').build(); insert acct;`. This is an advanced customization – initially, we assume basic explicit creation which is more universally understandable.

- **Minimal but Meaningful Data**: Each test should create the minimal set of records required to test the logic, but also reflect realistic scenarios. If the Apex method being tested calls another class or triggers an event, we should set data such that those paths execute (unless we explicitly want to isolate via stubbing). For example, if saving a Contact will fire a trigger that expects an Account with certain field, the test data factory or AI should ideally know to populate that. In general, the AI doesn't know the org's triggers, so the safest route is to populate standard required fields and perhaps some commonly expected fields (like an Account's Industry if the code often filters by it, etc.). We can instruct the AI to create data with obviously dummy but valid values (e.g., phone numbers "555-0101", emails "test@example.com", etc.) to make tests more illustrative. It's also a best practice *not to hard-code Ids* or sensitive data in tests. The AI will generate new records and use their runtime-generated Ids (storing them in variables as needed). If an external Id or specific record is needed (rare in isolated tests), the test should insert that record then use it.

- **Avoiding SeeAllData=True**: As a rule, we do not want the AI to use `@isTest(SeeAllData=true)` unless absolutely necessary (for example, testing a report or functionality that explicitly requires existing data like PriceBookEntries). Using existing org data can make tests flaky and non-isolated (Source: levelupsalesforce.com). Thus, the AI should default to SeeAllData=false (which is the default if not specified in API versions >= 24). We can have the generator explicitly mark tests with `@isTest` (without SeeAllData) to be clear. In cases where the code uses objects like Users or RecordTypes which are standard data but not accessible in tests by default, the correct approach is to either create required setup (e.g., create a user with profile as needed) or use provided objects (like `UserInfo.getUserId()` for running user, etc.). The tool may not delve that deep unless the class signature shows such needs (like a method parameter of type User or reference to a profile). Those scenarios might need user guidance to finalize.

- **Assertions and Outcome Checking**: A generated test isn't useful if it only runs the code without verifying results. Our AI will attempt to assert important outcomes. If the method returns a value (say, a List or a Boolean), the test should assert that value is as expected given the test inputs. If the method performs DML (e.g., updates a record), the test can query that record after the method call (within the test context) to verify the changes. For example, for a method that sets an Account's status, the test will insert an Account, call the method, then re-query that Account and `System.assertEquals(expectedStatus, acct.Status__c)`. We prompt the AI to include at least one assertion per test method, focusing on key fields or outputs. We also encourage it to use `System.assert` variants with messages for clarity, although Salesforce tests typically either use `System.assertEquals/NotEquals` or the newer `Test.assertEquals`. The AI likely knows the common usage. It should also assert that the size of returned collections or number of records created matches expectations. Essentially, we want the generated tests to not only increase coverage but also serve as **executable documentation** of the code's intended behavior.

- **DML Safety and Bulk Inserts**: When creating multiple records (e.g., testing bulk behavior or needing multiple related records), the AI will generate code to insert them in a single statement if possible. For example:

  apex

  Copy

  ```
  List<Contact> contacts = new List<Contact>{ new Contact(FirstName='Jim', LastName='Tester', AccountId=acct.Id), new Contact(FirstName='Pam', LastName='Tester', AccountId=acct.Id) }; insert contacts;
  ```

  This is better than two separate inserts. Similarly, if multiple object types need creation, order the DML to avoid hitting limits (and always insert parent objects like Account before child objects like Contact to satisfy lookup relations). Another safety measure is wrapping critical DML operations inside Test.startTest/stopTest if the code under test itself runs in context of DML (though typically that's used for resetting limits rather than mandatory for correctness).

- **Ensuring Unique Data**: Sometimes tests can interfere if data is not unique (though in isolation this is less of a concern). But if the code under test queries by some unique field (say looking up an Account by a unique name or external Id), the test should create a record with a unique value to avoid collisions or ensure it finds that record. The AI can be instructed to use a randomness or static counter in naming (e.g., "TestAccount1", "TestAccount2") if it's creating multiple accounts, ensuring uniqueness. Salesforce actually guarantees isolation, so collisions with existing org data aren't possible, but uniqueness can still matter logically within the test.

By embedding these best practices in either the LLM prompt or in post-processing rules, the generated tests will align with what seasoned Salesforce developers would write. This not only increases the likelihood that the test passes on first run, but also that other developers can trust and understand the test later. As Piotr Gajek noted, a proper test data factory or builder yields benefits like single-responsibility (each factory knows how to build a certain object) and easier maintenance when requirements change (Source: blog.beyondthecloud.dev). Our AI approach is essentially creating an *on-the-fly* test data factory tailored to each class under test, following similar principles. If down the line something in the data requirements changes (say a field becomes required), re-running the AI generator on the class could update the tests consistently.

Example Scenarios and AI-Generated Test Classes

To demonstrate how the AI-driven synthesizer works, let's walk through a couple of examples. We provide a sample Apex class signature as input and show the kind of test class the LLM would generate, with explanations.

Example 1: Service Class with SOQL Query and DML

Input Apex Class – Suppose we have an Apex class `ContactService` with a method that queries contacts for a given account and updates their status. The class might look like this (the AI would be given at least the signature, and ideally some docstring or comment about its behavior):

apex

Copy

```
public class ContactService { /** * Sets the Status__c of all Contacts for the given Account to "Active". */ public static void activateContacts(Id accountId) { List<Contact> contacts = [SELECT Id, Status__c FROM Contact WHERE AccountId = :accountId]; for (Contact c : contacts) { c.Status__c = 'Active'; } update contacts; } }
```

Here, the method `activateContacts` takes an Account Id and sets all related contacts' custom Status field to "Active". We need to test that it correctly updates contacts (and does nothing unexpected if there are no contacts).

AI-Generated Test Class – The tool, after processing the above, produces something like:

apex

Copy

```
@IsTest(seeAllData=false) private class ContactServiceTest { // Test data creation for all tests @testSetup static void setupData() { // Create a test Account and some Contacts for that Account Account testAcct = new Account(Name='Test Account'); insert testAcct; List<Contact> contacts = new List<Contact>{ new Contact(FirstName='Alice', LastName='Doe', AccountId=testAcct.Id, Status__c='Inactive'), new Contact(FirstName='Bob', LastName='Doe', AccountId=testAcct.Id, Status__c='Inactive') }; insert contacts; // Note: contacts[0] and contacts[1] now have Status__c = 'Inactive' } static testMethod void testActivateContacts_updatesStatus() { // Query the account and related contacts from setup Account acct = [SELECT Id FROM Account LIMIT 1]; List<Contact> contactsBefore = [SELECT Id, Status__c FROM Contact WHERE AccountId = :acct.Id]; System.assertEquals(2, contactsBefore.size(), 'There should be 2 contacts initially'); System.assertEquals('Inactive', contactsBefore[0].Status__c, 'Pre-condition: contact status is Inactive'); // Call the method under test Test.startTest(); ContactService.activateContacts(acct.Id); Test.stopTest(); // After invocation, all contacts for the account should have Status__c = 'Active' List<Contact> contactsAfter = [SELECT Id, Status__c FROM Contact WHERE AccountId = :acct.Id]; System.assertEquals(2, contactsAfter.size(), 'There should still be 2 contacts after activation'); for (Contact c :
```

```
contactsAfter) { System.assertEquals('Active', c.Status__c, 'Contact status should be Active after activation'); } } static testMethod void testActivateContacts_noContacts() { // Create a new Account with no contacts Account emptyAcct = new Account(Name='Empty Account'); insert emptyAcct; // Call the method on an account with no contacts (should not fail) Test.startTest(); ContactService.activateContacts(emptyAcct.Id); Test.stopTest(); // Verify that no error was thrown and nothing to assert (no contacts to check) // You might verify that it simply didn't crash. Optionally, confirm that zero contacts in DB remain zero. Integer contactCount = [SELECT count() FROM Contact WHERE AccountId = :emptyAcct.Id]; System.assertEquals(0, contactCount, 'No contacts should exist for the empty account'); } }
```

Explanation: This test class `ContactServiceTest` shows how the generator follows best practices and covers multiple scenarios:

- We see `@IsTest(seeAllData=false)` at the top, ensuring isolation. The class is marked private (typical for test classes).
- A `@testSetup` method creates common test data: one `Account` (`testAcct`) and two `Contact` records linked to it. This setup data is used by one of the test methods. The AI chose to create two contacts to simulate updating multiple records (bulk behavior) and set their `Status__c` initially to "Inactive" to later observe the change. It inserts them in a single DML operation (bulk insert) for efficiency.
- In `testActivateContacts_updatesStatus`, it queries the setup contacts (verifying the pre-condition). It then calls the target method within `Test.startTest()/Test.stopTest()`. After that, it queries again to verify the `Status__c` field on all contacts is now "Active". Multiple assertions are included: checking the count of contacts and their status values. This ensures the test will fail if the method didn't do its job for every contact.
- The test method name clearly indicates it's testing the update of status. The AI also generated a second test method: `testActivateContacts_noContacts`. This tests the edge case where the account has no contacts. It creates a new Account (separately, not using testSetup data) and calls `activateContacts` on it. The expectation is that nothing bad happens (no errors, and obviously no contacts to update). It verifies that the count of contacts for that account remains 0, essentially asserting the method didn't inadvertently insert or do anything weird. Including such an edge-case test shows the AI's capability to handle different logical branches (in this case, the branch where the query returns an empty list) – likely because the prompt or training encourages covering "zero records" scenario.

This example demonstrates the synthesizer's attention to details:

- **Required Fields**: It set `Account.Name` and `Contact.LastName` (the latter via LastName='Doe' for both contacts), which are required fields. Also set a dummy FirstName for completeness.

- **Meaningful Data**: It used "Inactive" and "Active" to mirror realistic status values, which makes the test more understandable (as opposed to some random strings).
- **Governor Compliance**: The queries and inserts are well within limits (only a couple of SOQLs and DMLs). The use of testSetup ensures the main test doesn't do repeated inserts for the same base data.
- **Multiple Scenarios**: By generating two test methods, it increases coverage (both when contacts exist and when none exist). If the class had more methods, it would generate corresponding tests for each.

The developer could now review this output. Perhaps they notice that in the second test, there is no explicit assertion of "no error". In Apex, if an unhandled exception was thrown by `activateContacts`, the test would fail, so implicitly that test covers the no-error scenario. The comment is enough. The developer might accept this as is.

If they wanted to refine (say the requirement was that if no contacts, maybe it should throw an exception – but in our class it doesn't), they could modify accordingly. But assuming this matches intent, the test can be saved and run, immediately giving us coverage on `ContactService.activateContacts` with minimal manal effort.

Example 2: Custom Object with Calculation Logic

Input Apex Class – Consider a custom scenario with two custom objects, `Invoice__c` and `Line_Item__c`. Invoices have line items, and we want to calculate the total amount of an invoice. The Apex class might be:

apex

Copy

```
public class InvoiceCalculator { // Computes total Price for an invoice by summing its line items public static Decimal calculateTotal(Invoice__c inv) { Decimal total = 0; for (Line_Item__c item : [SELECT Quantity__c, Unit_Price__c FROM Line_Item__c WHERE Invoice__c = :inv.Id]) { total += (item.Quantity__c == null ? 0 : item.Quantity__c) * (item.Unit_Price__c == null ? 0 : item.Unit_Price__c); } return total; } }
```

The method `calculateTotal` takes an `Invoice__c` record (could be not inserted yet, but here presumably an invoice Id is used in query) and sums up Quantity * Unit_Price for each related line item. We need to test that it correctly sums values and handles nulls.

AI-Generated Test Class – The output might be:

apex

Copy

```
@isTest private class InvoiceCalculatorTest { static testMethod void testCalculateTotal_basicSum() { // Create an Invoice record Invoice__c inv = new Invoice__c(Name='Test Invoice'); insert inv; // Create two Line_Item__c records related to this invoice Line_Item__c item1 = new Line_Item__c(Invoice__c = inv.Id, Quantity__c = 2, Unit_Price__c = 50); Line_Item__c item2 = new Line_Item__c(Invoice__c = inv.Id, Quantity__c = 1, Unit_Price__c = 100); insert new Line_Item__c[]{ item1, item2 }; // Call the calculator Test.startTest(); Decimal total = InvoiceCalculator.calculateTotal(inv); Test.stopTest(); // The expected total is 2*50 + 1*100 = 200 System.assertEquals(200, total, 'Total should be sum of line item prices'); } static testMethod void testCalculateTotal_handlesNulls() { // Create an Invoice with line items that have null values Invoice__c inv2 = new Invoice__c(Name='Test Invoice Nulls'); insert inv2; Line_Item__c itemA = new Line_Item__c(Invoice__c = inv2.Id, Quantity__c = 3, Unit_Price__c = null); Line_Item__c itemB = new Line_Item__c(Invoice__c = inv2.Id, Quantity__c = null, Unit_Price__c = 10); insert new Line_Item__c[]{ itemA, itemB }; Test.startTest(); Decimal total2 = InvoiceCalculator.calculateTotal(inv2); Test.stopTest(); // Expected: itemA contributes 3*0 = 0 (Unit_Price null treated as 0), itemB contributes 0*10 = 0, total = 0. System.assertEquals(0, total2, 'Total should treat null quantities or prices as 0'); } }
```

Explanation: In this test, because the logic is a pure function (no state changes in the database except the query), using `Test.startTest()` may not be strictly necessary for functionality – but it's still used to simulate best practice isolation of the execution. The test focuses on validating the returned value.

- The first test method `testCalculateTotal_basicSum` creates an `Invoice__c` (note: custom objects require a Name as well, which is provided as 'Test Invoice'). It inserts it to get an Id, because the code uses `inv.Id` in the query. Then it creates two `Line_Item__c` records linked to that invoice Id, with specific Quantity and Unit_Price values (2 @ 50, and 1 @ 100). After inserting them, it calls `calculateTotal(inv)`. The returned Decimal is asserted to equal 200. This checks that the summation logic works for normal non-null values.

- The second test, `testCalculateTotal_handlesNulls`, addresses a subtle aspect: the code treats null Quantity or Price as 0. It creates an invoice and two line items where one has `Unit_Price__c = null` and the other has `Quantity__c = null`. After insertion, calling the method should yield 0 (since effectively 3_0 + 0_10 = 0). The test asserts that. This ensures that if the code didn't handle nulls properly, the test would catch it. This test was likely generated because the AI recognized from the code that null handling is a branch (the ternary `== null ? 0 : value`). The AI thus invented a scenario to cover it. This is a good example of AI going beyond a trivial test: it not only covers the general case but also an edge case which a human tester would consider.

We also observe:

- It uses list initialization for inserting multiple line items at once, again an efficient practice.

- It gave distinct names to invoices to avoid any confusion (though not strictly necessary in isolation, but good for readability).
- It doesn't use a shared `@testSetup` here; possibly because each test uses distinct data and there's no repeated setup to factor out. This is fine; using testSetup for only one method's data doesn't add value.

The output is a clean, focused test class. It leverages the knowledge of custom object fields `Quantity__c` and `Unit_Price__c` (which we assume were provided via field definitions in the prompt). The tool might have known these from a snippet of the class or an accompanying description of the custom object. In practice, we could retrieve the Custom Object's fields via Tooling API to inform the AI. Ensuring the AI knows the correct API names (like `Unit_Price__c`) is crucial – if the prompt had a typo or the AI hallucinated a different field name, the test would not compile. By feeding the actual field names (perhaps from the class's SOQL string, which the AI can see, or from a describe call), we anchor it in reality.

Both examples show the **value of AI**: it created non-obvious tests (the empty contacts scenario, the null handling scenario) that cover more than just the "happy path". These are the kinds of tests that improve confidence in code quality. A deterministic template might not have thought to do that without explicit programming. The AI leveraged the logic present in code to suggest those cases – a sign of understanding the intent to some degree.

Integration with Salesforce APIs and Developer Tools

To implement this solution in the real Salesforce ecosystem, we need to integrate with various tools and APIs. Here we outline how the test synthesizer interacts with Salesforce and developer tooling:

- **Salesforce DX (CLI) Integration**: The Salesforce CLI (`sfdx`) is a primary tool for fetching and deploying metadata in a developer workflow. Our AI generator can be built as a plugin or script that works alongside the CLI. For example, a developer could run a command like `sfdx apexgen:testdata -c MyClass.cls` which behind the scenes:
  1. Retrieves `MyClass.cls` metadata (if not already in the local project) using `sfdx force:source:retrieve` or by reading the local file.
  2. Calls the AI service with the class signature and possibly additional context.
  3. Receives the test code and saves it as `MyClassTest.cls` in the project.
  4. Optionally, immediately runs `sfdx force:source:deploy` (or `force:apex:test:run`) to deploy and execute the test, verifying it passes.

By automating these steps, the developer can seamlessly incorporate AI generation into their development cycle. CLI integration also means it can be used in CI pipelines (for example, auto-generate tests for new classes in a pull request).

- **VS Code Extension**: Salesforce provides an official VS Code extension pack. The Agentforce feature is likely part of that (with a UI for test generation as described (Source: developer.salesforce.com)). We could either hook into that or create our own extension. A VS Code extension could provide a context menu "Generate Test Class (AI)" when right-clicking an Apex class. This would trigger our backend process. The advantage of a VS Code integration is that we can show a diff or preview of the generated test and perhaps allow the user to edit it in the editor with AI suggestions (like GitHub Copilot might). Given the collaborative nature of LLMs, such an extension could even allow chat-based refinement: e.g., open a panel where the developer can type, "Add another test method for when parameter X is null" and the AI then modifies the code accordingly.

- **Metadata API**: For retrieving and deploying classes, the Metadata API is fundamental. It has operations to retrieve components (like classes, objects) either via package.xml definitions or direct API calls. Tools like the SalesforceBen Metadata API guide (Source: salesforceben.com) show how you can fetch a class's content. Internally, Salesforce's Apex provides a `Metadata.Operations.retrieve` in Apex itself (Source: salesforcebuddy.com) (though that's more for use inside Apex, which is not our case). We will likely use either the JS Force library (for Node) or a Python simple-salesforce library to call the Metadata API from our tool, if not using CLI. After generation, deploying the test class uses `Metadata.deploy` with the class file packaged in a zip. Alternatively, since test classes are Apex, we could use the **Tooling API** to create it: the Tooling API has a resource to insert an `ApexClass` sObject (with fields like Name, Body). If using an OAuth connection, posting to `/services/data/vXX.X/tooling/sobjects/ApexClass` with JSON containing the body might directly create the class in the org (though it might bypass some validations that Metadata deploy would catch, so Metadata API is safer).

- **Tooling API and SymbolTable**: The Tooling API can retrieve an Apex class's symbol table, which is a JSON listing of its methods, properties, and references. This could be useful to programmatically determine which SObject types the class references (e.g., if symbol table shows it uses `Account` or custom object types). It might also list if it implements an interface (which could hint needing a stub in test) or if it's a trigger handler requiring certain records. Using this info, the prompt to the AI can be enriched (for instance: "Class MyClass references Contact, Account objects and makes an HTTP callout" – so AI knows to include contact/account data and maybe a callout mock). We can obtain a symbol table by querying Tooling API: `SELECT SymbolTable from ApexClass WHERE Name='MyClass'`. The symbol table contains e.g. an array of `externalReferences` with names of sObjects or classes referenced. This mechanistic approach ensures we don't miss a needed object's data.

- **Schema Describe (SOAP/REST)**: In addition to or instead of Tooling API for fields, we can call the Salesforce REST API (with the tooling or metadata scope) to get object describes. For example, a GET to `/services/data/v57.0/sobjects/Account/describe` returns JSON of all Account fields, including

which are required (`nillable=false` and `defaultedOnCreate=false` typically means required), data types, length, etc. For custom objects or fields, the same works (`.../sobjects/Invoice__c/describe`). This is simpler than parsing the XML from Metadata API for a CustomObject. Using this, we can dynamically build a snippet for the prompt like: "Account required fields: Name; Contact required fields: LastName; Invoice__c required: Name; Line_Item__c required: (none beyond lookup to Invoice)". The AI can use this to avoid omissions. This approach requires the tool to have API access (likely via OAuth session or using the user's sfdx auth).

- **Einstein GPT and Apex**: It's worth noting Salesforce's own **Einstein GPT** for Developers might provide an official API or interface to do some of this in the future. As of mid-2025, Salesforce announced things like Apex GPT integration (Agentforce) with presumably some fine-tuning on Apex. Our tool could either call OpenAI's API or potentially a Salesforce-provided LLM endpoint (ensuring data stays within Salesforce's trust boundary if that's a concern). If using OpenAI or Azure OpenAI, we must consider data security (source code and schema being sent to an external service). One could anonymize certain identifiers in the prompt (e.g., remove org-specific names) if needed, or opt for an on-premise model as discussed in the next section.

- **Testing the Test Generator**: Finally, the irony – we should test our generator. One could imagine having Apex classes as input and expected test classes as output for known cases, to evaluate the AI's performance. This is more on the development of the tool itself, but it's something to consider (perhaps using smaller prompt+completion for verification or using static analysis to verify the generated test indeed covers the methods intended). For instance, after generation, the tool could quickly parse the test class to ensure every public method of original class is called at least once in the test. If not, it might warn the user or attempt another generation.

Model Selection, Fine-Tuning, and Deployment Options

A critical aspect of this system is the AI model generating the Apex code. We have choices in model selection and how to host it, each with implications:

- **Use of Large Pre-trained Models (GPT-4, etc.)**: Models like OpenAI's GPT-4 are very powerful, having been trained on massive code datasets (hundreds of billions of parameters, including likely Apex-like languages or enough general knowledge to adapt) (Source: salesforcedevops.net). They exhibit *emergent abilities* to understand context and generate coherent, contextually appropriate code. Using GPT-4 via OpenAI's API (or Azure's hosted instance) would likely yield the highest quality output currently – it can handle nuanced instructions and produce correct Apex syntax in most cases. The advantage is we don't have to train a model from scratch. The disadvantage is sending potentially sensitive code or schema data to an external service, and relying on a third-party service with per-request cost. There's also rate limiting and the need for internet connectivity from the dev environment (though VS Code with internet or a server proxy can manage that).

- **Salesforce Einstein GPT (Code)**: Salesforce might offer their own version of code generation integrated with Salesforce data. If such an API is accessible, it could be an option, possibly ensuring that the data stays within Salesforce's servers. It could even have some Salesforce-specific tuning. However, details on that are scant publicly beyond the VS Code plugin.

- **Fine-Tuned Code LLM on Apex**: Another approach is to fine-tune an existing open-source LLM on a corpus of Apex code. There are code-specialized models (like Salesforce's CodeGen series, HuggingFace's StarCoder, Meta's LLaMA 2 with coding instruction tuning, etc.). Fine-tuning means taking a base model and training it further on domain-specific data. If a company had a large repository of Apex classes and test classes, in theory one could fine-tune a model to learn the mapping from class to test. This could make it better at Apex-specific idioms and possibly more deterministic in output for that company's coding standards. Fine-tuning can be done on models like CodeT5, Codex (if OpenAI allowed, but they pivot more to instructions rather than fine-tuning now), or smaller LLMs that one can run on-premise. The benefits are:

  - The model might pick up on common patterns (e.g., how that org usually structures tests, or naming conventions).
  - The model can be hosted internally to alleviate privacy concerns.
  - Once fine-tuned, generation might be faster for that domain (and possibly cheaper if running on owned hardware).

  The downsides:

  - Fine-tuning requires a substantial dataset and expertise. Apex code corpora are not as readily available as, say, Java or Python. One might extract open-source packages or ask the community for contributions, but many Apex codebases are proprietary. There is some public code on GitHub (e.g., Salesforce sample apps, triggers, frameworks) that could form a base.
  - Even if fine-tuned, the model may not reach GPT-4's level of "general intelligence" in understanding requirements.
  - Maintaining the model (updates, ensuring it generalizes to new patterns) is non-trivial.

  As an intermediate step, one could use **prompt fine-tuning** (few-shot learning): i.e., include one or two example class-to-test transformations in the prompt as a guide. This often boosts output quality without actual model training. For instance, prompt: "Example:\n\nTest:\n\nNow generate test for the following class:\n".

- **Local vs Hosted Deployment**: Table 2 compares these options:

MODEL DEPLOYMENT	DESCRIPTION	PROS	CONS
Cloud-hosted LLM (OpenAI GPT-4, Azure OpenAI, etc.)	Leverage a third-party API to generate code. GPT-4 is nearly 1 trillion parameters:contentReference\[oaicite:43\]{index=43}, with broad knowledge and capability.	- State-of-the-art performance on code generation (high quality) - No need to manage infrastructure or model updates - Scales on demand with high availability	- Code and metadata sent off-site (potential IP/security concerns):contentReference\[oaicite:44\]{index=44} - Recurring usage cost (API calls) - Dependent on external service uptime and latency
On-Premise / Self-Hosted LLM (Fine-tuned open-source model deployed in-house)	Run the AI model on the company's own servers or developer machine. E.g., a 13B parameter model fine-tuned on Apex code.	- Data stays within the organization (improved privacy):contentReference\[oaicite:45\]{index=45} - Can customize model (fine-tune) to org-specific code style - Potentially lower variable cost if infrastructure is in place (after initial setup)	- Requires powerful hardware (GPUs) and ML expertise to deploy - May not match quality of the largest proprietary models - Longer iteration to update model with new data or improvements

In many enterprise settings, data security is paramount, so a popular approach is a **hybrid**: use a local model for anything involving sensitive code, and use a cloud model for more generic tasks. For our test generator, a company might start with GPT-4 during prototyping (to maximize success in generation) and later transition to a smaller local model that has been bootstrapped with knowledge gained (e.g., by fine-tuning on outputs from GPT-4 on a large sample of classes – effectively distilling knowledge). We should highlight that any use of production code in prompts should be done carefully in compliance with company policy and possibly with OpenAI's data usage policies (OpenAI allows opting out of data retention, etc., which one would do). The reference to one Reddit user's "Custom GPT for Apex Test Class Generator" suggests community interest in using GPT models for exactly this task (Source: reddit.com), indicating that hosted models are already being tried by Salesforce devs.

Finally, whichever model is used, we must handle **model limitations** – LLMs sometimes produce output that looks correct but isn't (logic errors or minor syntax issues). Thus, integrating a validation step is wise: after generation, we could run the Apex Compiler (via the Tooling API or sfdx force:apex:compile if such existed) to check syntax. Salesforce doesn't have a standalone compiler API, but deploying to a scratch org will compile it. If compilation fails, we catch the errors and potentially feed them back to the AI (e.g., "the code didn't compile, error on line X: Unknown field Y, please fix that"). This turns into another prompt to correct mistakes. This automated feedback loop can vastly improve reliability. It's similar to how Copilot might suggest code and the developer sees a red squiggly and either edits or asks for a fix.

In summary, the model choice is a balance between **quality, privacy, and cost**. Many will opt for GPT-4 or successors for quality, at least initially. Salesforce's own offerings might soon provide high-quality codegen with more integrated privacy (e.g., running on Salesforce's trusted cloud). Over time, as open-source LLMs improve, an on-prem solution fine-tuned to Apex could become viable and cost-effective for large teams.

Conclusion and Future Outlook

Building an AI-driven test data synthesizer for Apex tests stands to significantly enhance developer productivity on the Salesforce platform. By automating the rote parts of test writing – especially constructing and inserting test records – developers can focus on verifying business logic rather than wrestling with setup. The proposed system combines Salesforce metadata intelligence with the generative power of LLMs to produce test classes that are not only covering code, but are meaningful and maintainable.

We have outlined how such a tool can be architected: from retrieving class signatures and schema via Salesforce APIs, to prompting an LLM (like GPT) to generate the test code, and integrating the result back into the development workflow. The approach respects Salesforce's unique constraints (governor limits, isolation, etc.) and follows best practices in test data creation (ensuring required fields, minimal necessary data, clear assertions). The examples provided illustrate that an AI can write competent Apex tests for both standard scenarios and edge cases, sometimes catching things a human might miss. This not only saves time but can improve code quality by encouraging more thorough testing.

In weighing prompt-based generation versus deterministic templating, we acknowledge the need for consistency and reliability in enterprise environments. Our design tries to get the best of both: leveraging AI's flexibility while keeping the outputs in check through user review, iterative prompts, and possibly combining templates for structure. As the technology matures, we might see more deterministic behavior from specialized models or hybrid systems (for example, first using static analysis to outline test cases, then AI to fill in the guts).

Looking forward, several opportunities could make this even more powerful:

- **Deeper Static Analysis**: The tool could analyze the Apex code to derive expected outcomes (like symbolic execution) and have the AI assert not just basic outcomes but correct business logic results. This could bring the generated tests closer to true specification-based tests, not just code coverage.
- **Integration with ALM**: Tying test generation into source control or CI. For instance, if a pull request lacks tests for new Apex classes, the CI could auto-generate draft tests and include them for the reviewer to consider – jumpstarting the review process.
- **User Feedback Loop**: Incorporate feedback from actual test runs. If a generated test fails, the failure message could be fed to the AI to improve the test (e.g., adjust an assertion or add handling for an unexpected outcome).

- **Broader AI Assistants**: Expand beyond test classes – perhaps generate test data creation scripts for manual QA, or documentation of what scenarios the tests cover, etc. With Salesforce's increasing AI offerings, one can imagine a future DevOps process where an AI agent monitors code and suggests tests continuously (Salesforce has hinted at "1 million Agentforce conversations" and evolving AI in their ecosystem (Source: salesforcedevops.net)).

In conclusion, an AI-driven Apex test data synthesizer exemplifies the positive impact of generative AI on software engineering: reducing drudgery, enforcing best practices, and ultimately leading to more robust applications. By carefully blending Salesforce platform knowledge with modern AI models, enterprise developers and architects can achieve a new level of efficiency in meeting the always-important requirement of quality (and 75% coverage!) in their Salesforce deployments. The implementation details and examples given in this report can serve as a blueprint for teams interested in bringing this capability to their development process today. With the right safeguards and iterative improvements, AI-generated tests can become as trustworthy as handwritten ones – and a lot faster to produce.

Sources:

1. Salesforce Developers – *Apex Test Case Generation (Agentforce)*(Source: developer.salesforce.com) (Source: developer.salesforce.com)
2. Salesforce Developer Blog – Vernon Keenan, *How To Use ChatGPT for Salesforce Generative Coding*(Source: salesforcedevops.net)(Source: salesforcedevops.net)
3. Salesforce Ben – *What Are Salesforce Governor Limits?*(Source: salesforceben.com)
4. Diffblue Post – Zoe Laycock, *Why Deterministic Test Generation Is Important*(Source: diffblue.com) (Source: diffblue.com) (Source: diffblue.com)
5. Beyond the Cloud Blog – Piotr Gajek, *Apex Test Data Factory*(Source: blog.beyondthecloud.dev)(Source: blog.beyondthecloud.dev)
6. SoftServe Blog – *Speed up Unit Tests using Stub API* (on avoiding DML in tests) (Source: softserveinc.com)(Source: softserveinc.com)
7. Salesforce StackExchange – *Do we get new governor limits in test classes for each test method?*(Source: salesforce.stackexchange.com) (accessed via search)
8. Salesforce Developers Documentation – *Isolation of Test Data from Org Data*(Source: levelupsalesforce.com) (via summary)
9. Reddit – *Custom GPT for Salesforce: Apex Test Class Generator* (user discussion) (Source: reddit.com) (indirect reference)

10. SalesforceDevops.net – *Generative Coding Takes Off*(Source: salesforcedevops.net) (AI limitations and privacy)

Tags: salesforce, apex, apex unit testing, ai, large language models, code generation, test data automation

About Cirra

About Cirra AI

Cirra AI is a specialist software company dedicated to reinventing Salesforce administration and delivery through autonomous, domain-specific AI agents. From its headquarters in the heart of Silicon Valley, the team has built the **Cirra Change Agent** platform—an intelligent copilot that plans, executes, and documents multi-step Salesforce configuration tasks from a single plain-language prompt. The product combines a large-language-model reasoning core with deep Salesforce-metadata intelligence, giving revenue-operations and consulting teams the ability to implement high-impact changes in minutes instead of days while maintaining full governance and audit trails.

Cirra AI's mission is to **"let humans focus on design and strategy while software handles the clicks."** To achieve that, the company develops a family of agentic services that slot into every phase of the change-management lifecycle:

- **Requirements capture & solution design** – a conversational assistant that translates business requirements into technically valid design blueprints.
- **Automated configuration & deployment** – the Change Agent executes the blueprint across sandboxes and production, generating test data and rollback plans along the way.
- **Continuous compliance & optimisation** – built-in scanners surface unused fields, mis-configured sharing models, and technical-debt hot-spots, with one-click remediation suggestions.
- **Partner enablement programme** – a lightweight SDK and revenue-share model that lets Salesforce SIs embed Cirra agents inside their own delivery toolchains.

This agent-driven approach addresses three chronic pain points in the Salesforce ecosystem: (1) the high cost of manual administration, (2) the backlog created by scarce expert capacity, and (3) the operational risk of unscripted, undocumented changes. Early adopter studies show time-on-task reductions of 70-90 percent for routine configuration work and a measurable drop in post-deployment defects.

Leadership

Cirra AI was co-founded in 2024 by **Jelle van Geuns**, a Dutch-born engineer, serial entrepreneur, and 10-year Salesforce-ecosystem veteran. Before Cirra, Jelle bootstrapped **Decisions on Demand**, an AppExchange ISV whose rules-based lead-routing engine is used by multiple Fortune 500 companies. Under his stewardship the firm reached seven-figure ARR without external funding, demonstrating a knack for pairing deep technical innovation with pragmatic go-to-market execution.

Jelle began his career at ILOG (later IBM), where he managed global solution-delivery teams and honed his expertise in enterprise optimisation and AI-driven decisioning. He holds an M.Sc. in Computer Science from Delft University of Technology and has lectured widely on low-code automation, AI safety, and DevOps for SaaS platforms. A frequent

podcast guest and conference speaker, he is recognised for advocating "human-in-the-loop autonomy"—the principle that AI should accelerate experts, not replace them.

Why Cirra AI matters

- **Deep vertical focus** – Unlike horizontal GPT plug-ins, Cirra's models are fine-tuned on billions of anonymised metadata relationships and declarative patterns unique to Salesforce. The result is context-aware guidance that respects org-specific constraints, naming conventions, and compliance rules out-of-the-box.
- **Enterprise-grade architecture** – The platform is built on a zero-trust design, with isolated execution sandboxes, encrypted transient memory, and SOC 2-compliant audit logging—a critical requirement for regulated industries adopting generative AI.
- **Partner-centric ecosystem** – Consulting firms leverage Cirra to scale senior architect expertise across junior delivery teams, unlocking new fixed-fee service lines without increasing headcount.
- **Road-map acceleration** – By eliminating up to 80 percent of clickwork, customers can redirect scarce admin capacity toward strategic initiatives such as Revenue Cloud migrations, CPQ refactors, or data-model rationalisation.

Future outlook

Cirra AI continues to expand its agent portfolio with domain packs for Industries Cloud, Flow Orchestration, and MuleSoft automation, while an open API (beta) will let ISVs invoke the same reasoning engine inside custom UX extensions. Strategic partnerships with leading SIs, tooling vendors, and academic AI-safety labs position the company to become the de-facto orchestration layer for safe, large-scale change management across the Salesforce universe. By combining rigorous engineering, relentlessly customer-centric design, and a clear ethical stance on AI governance, Cirra AI is charting a pragmatic path toward an autonomous yet accountable future for enterprise SaaS operations.
