Mobile Applications

8. Test automation in Android

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2025/2026

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

- In this course, we have studied the fundamentals of Android development
- To verify that an application works as expected, developers typically run it on an Android Virtual Device (AVD) or a physical device and interact with it manually
- This process is known as **manual testing** and is a common part of everyday development
- However, in professional software development, **automated testing** is also essential to validate behavior efficiently and reliably

Table of contents

2. Software testing

- **Software testing** is the dynamic evaluation of a software system, called the System Under Test (SUT), to check whether it behaves as expected and satisfies user needs
- We can distinguish two main categories of software testing:

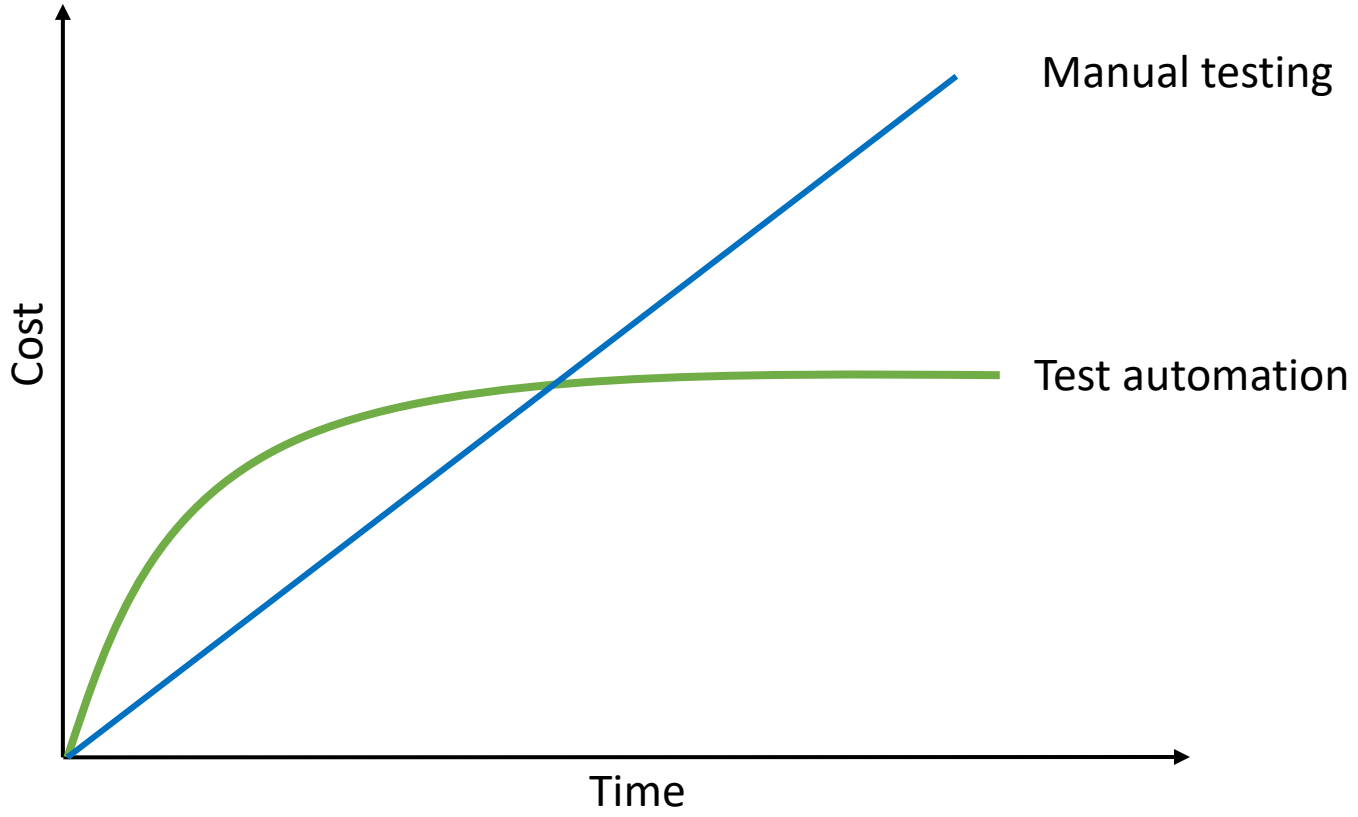
1. **Manual testing**. A person, such as a developer, tester, or end user, interacts with the SUT and evaluates its behavior
2. **Automated testing**. A developer or tester writes executable test code, called a **test case** (or simply a test), and uses software tools to run it against the SUT automatically

2. Software testing - Test automation

- The main benefits of **test automation** are the following:
  - Early defect detection: helps identify *bugs* earlier in the development process
  - Faster feedback: tests run quickly and reveal whether recent code changes introduced problems
    - A regression is a defect introduced into functionality that previously worked correctly
  - Repeatability: tests can be executed consistently with the same steps and expected results
  - Scalability: tests can be run across multiple devices, platforms, and environments
  - Efficiency: although automation requires an initial investment, it reduces repetitive manual effort and lowers long-term cost

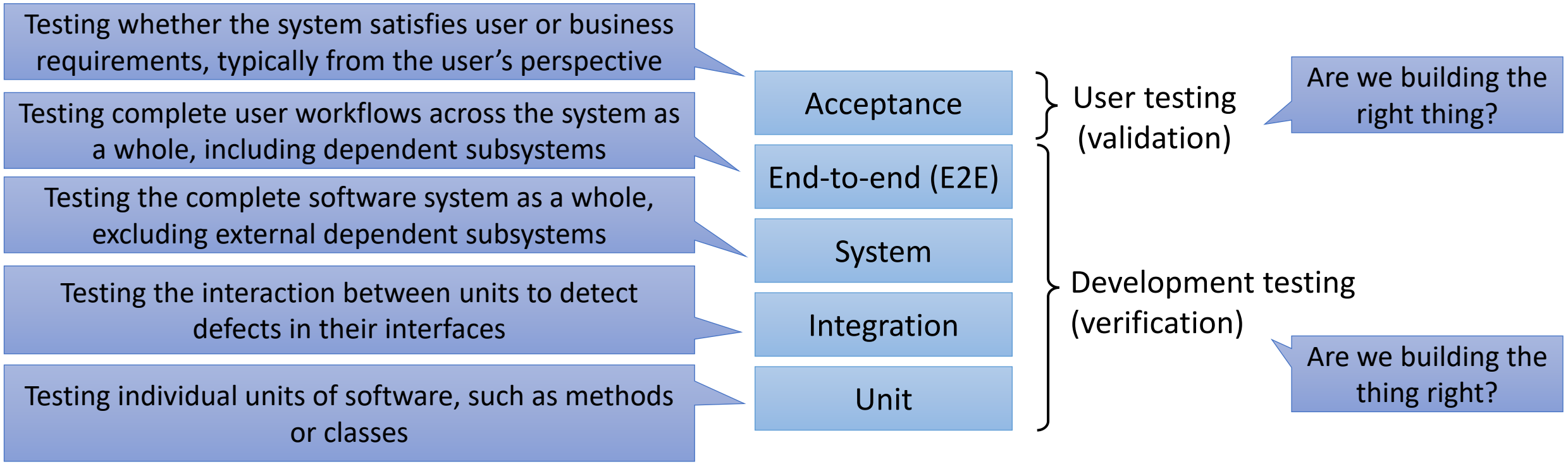
2. Software testing - Test automation

Test automation usually involves a higher initial cost, but it becomes more cost-effective than manual testing in the long term

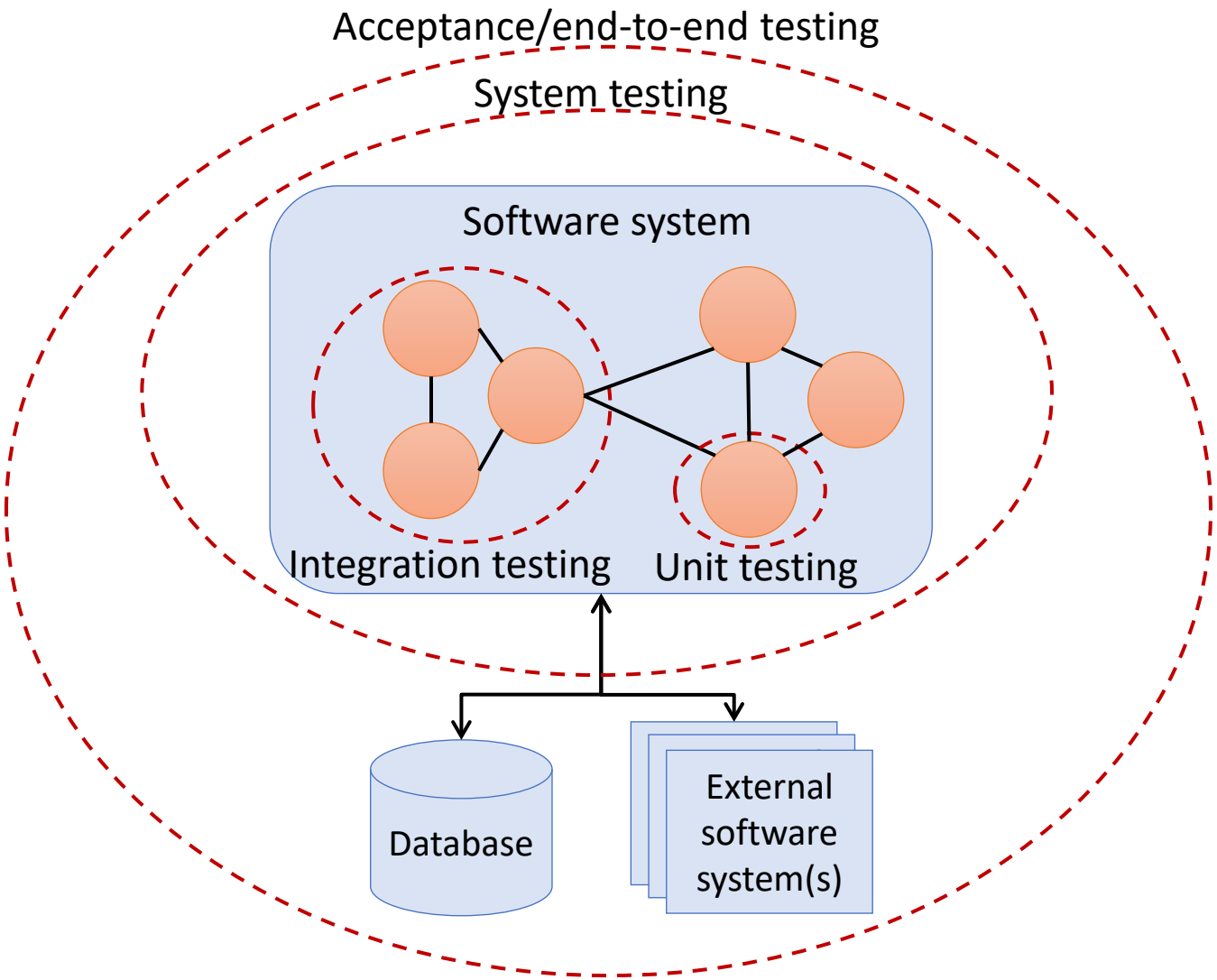


2. Software testing - Levels of testing

- Depending on the scope of the **System Under Test** (SUT), we can distinguish different levels of testing
  - These levels help software teams organize their testing activities according to how much of the system is being evaluated



2. Software testing - Levels of testing



2. Software testing - Types of testing

- Depending on the strategy for designing test cases, we can implement different types of tests:

1. **Functional testing** (black-box): verifies that the app behaves as expected
   - Example: when the user taps *"Add to cart"*, the selected item is added to the shopping cart
2. **Structural testing** (white-box): verifies the internal logic of the code
   - Example: a unit test checks that the `calculatePrice()` method returns the correct result in different situations
3. **Nonfunctional testing**: verifies quality attributes of the system, such as performance, security, usability, accessibility, etc.
   - Example: the app should load a screen in less than 2 seconds

Table of contents

3. Test automation tools

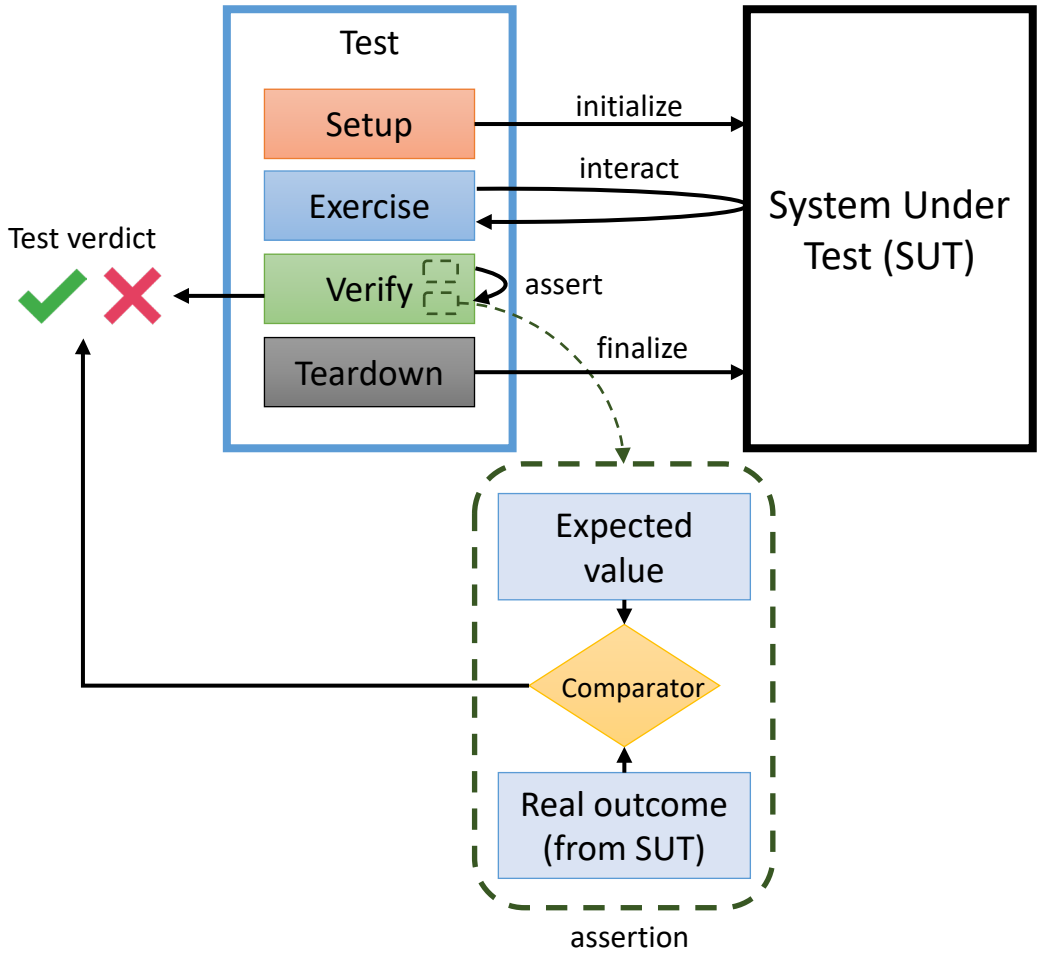
- Automated testing requires **tools** to implement, execute, and manage tests effectively
- One of the most important categories of testing tools is **unit testing frameworks**
- A unit testing framework allows developers to write, organize, and run repeatable tests

Library vs framework:

- A library is a collection of reusable code that developers call to solve specific problems
- A framework is a broader structure that combines libraries, tools, and conventions to support software development
- In general, a framework provides more structure and imposes more rules than a library

3. Test automation tools - Unit tests

- A unit test usually follows four phases: setup, exercise, verify, and teardown



3. Test automation tools - Unit tests

- In Java/Kotlin, the most relevant unit testing frameworks are:

https://junit.org/

https://testng.org/

Android Studio uses JUnit 4 by default

3. Test automation tools - E2E tests for mobile apps

- Mobile apps require specific tools to automate E2E testing
- Common tools for Android and mobile testing include:

https://developer.android.com/training/testing/espresso

https://appium.io/

In this course, we focus on the specific testing libraries for Jetpack Compose apps

https://developer.android.com/develop/ui/compose/testing

3. Test automation tools - Compose testing

- Compose provides testing APIs to find UI elements, perform actions, and verify results

1. Finders	2. Matchers	3. Actions
4. Assertions	5. Rules	6. Debug

The Compose Testing Cheat Sheet is a quick reference for the most useful test APIs

https://developer.android.com/develop/ui/compose/testing/testing-cheatsheet

3. Test automation tools - Compose testing

- Finders: locate an element
  - *onNodeWithText*(...)
  - *onNodeWithTag*(...)
  - *onNodeWithContentDescription*(...)
- Matchers: refine how elements are located
  - *hasText*(...)
  - *hasTestTag*(...)
  - *isEnabled*()
- Actions: interact with an element
  - *performClick*()
  - *performTextInput*(...)
  - *performScrollTo*()

3. Test automation tools - Compose testing

- Assertions: verify an element
  - `assertExists()`
  - `assertIsDisplayed()`
  - `assertTextEquals(...)`
  - `assertIsEnabled()`
- Rules: provide the test environment
  - `createAndroidComposeRule<Activity>()` (used when the test needs an Android activity)
- Debug: inspect problems
  - `printToLog()`
  - `printToString()`

Table of contents

4. Automated tests in Android

- Android Studio integrates with the main tools used for automated testing in Android. In this course, we use:
  - JUnit for unit testing
  - Compose testing libraries for UI testing
- The required testing dependencies, typically included by default in new Android Studio projects, are:

build.gradle.kts (app)

```
testImplementation(libs.junit)
androidTestImplementation(platform(libs.androidx.compose.bom))
androidTestImplementation(libs.androidx.ui.test.junit4)
```

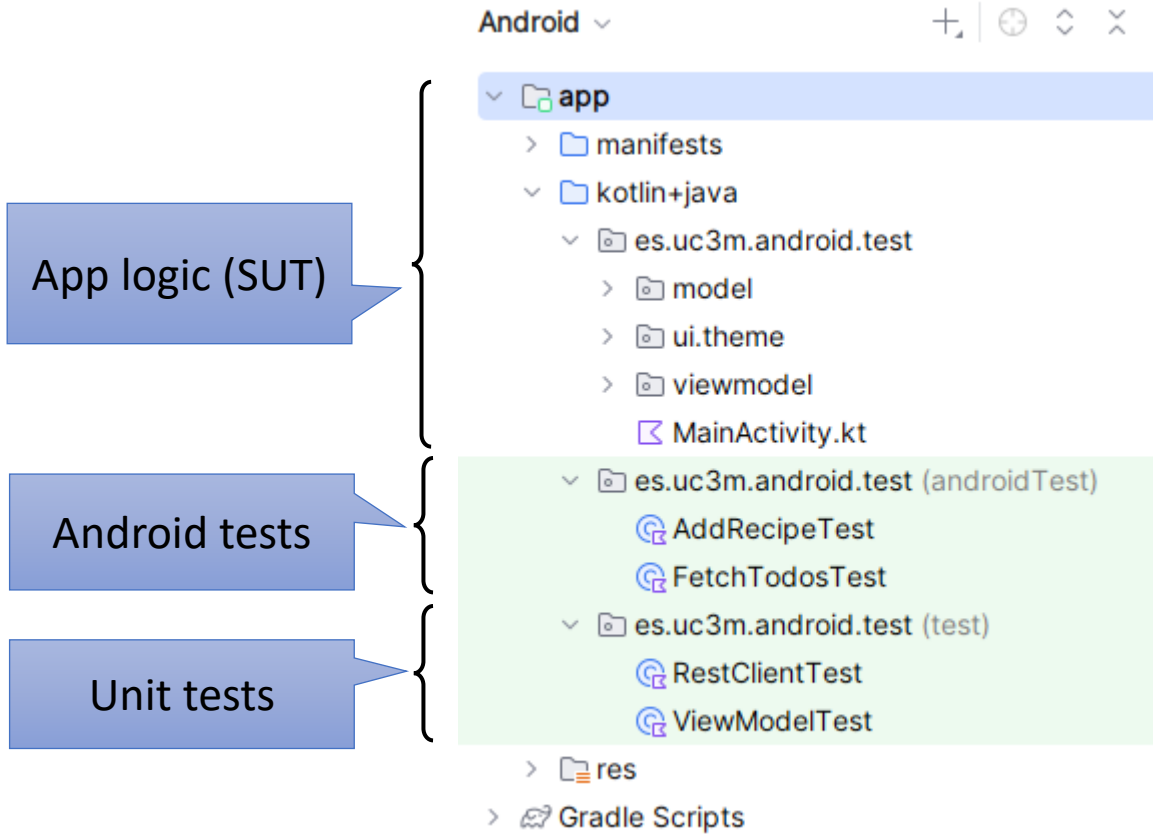
libs.version.toml

```
[versions]
junit = "4.13.2"
composeBom = "2026.03.00"

[libraries]
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit4" }
```

https://developer.android.com/studio/test
https://developer.android.com/training/testing

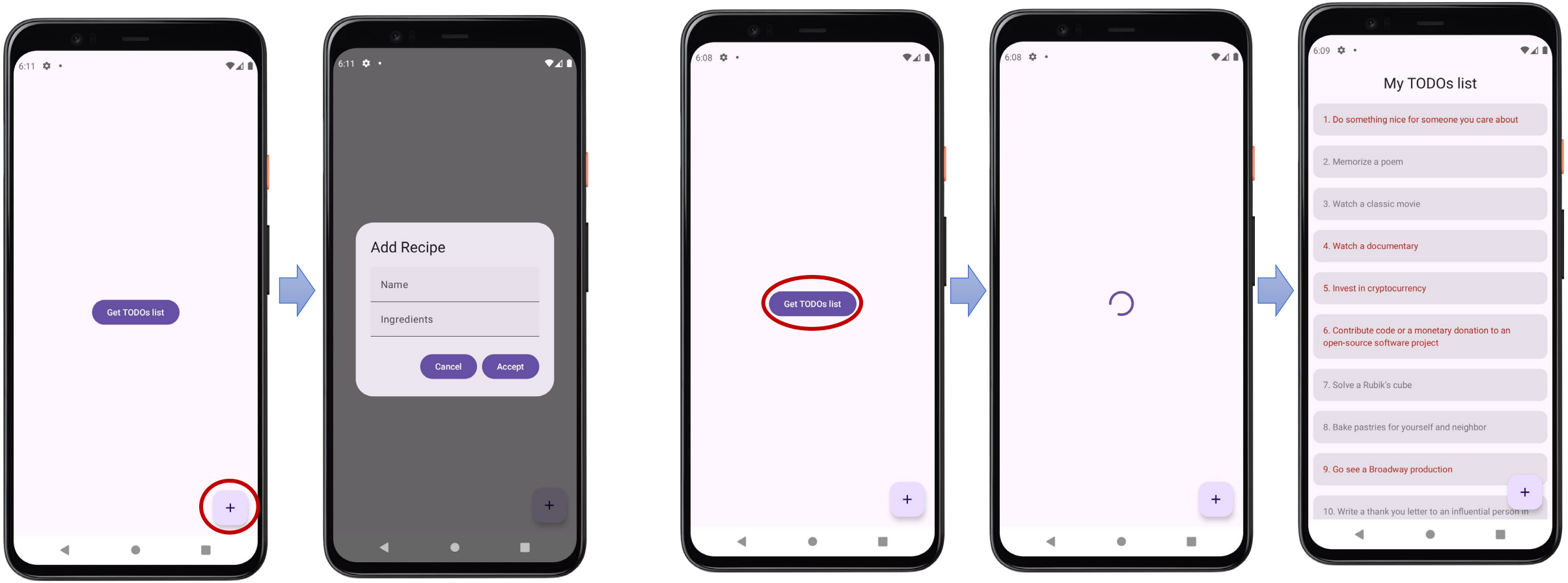
4. Automated tests in Android

- Android makes the following distinction between tests:
  - **Unit tests** (also called *local* tests): Small and fast, isolated
  - **Android tests** (also called *instrumented* tests): run on an Android device, either physical or emulated. There are two types:
    - UI tests (E2E tests): verify app behavior by interacting with the UI
    - Context tests (integration tests): verify behavior that depends on Android components such as the `context` object



4. Automated tests in Android - Android tests

- In the following examples, we use the demo app implementing a REST client as SUT:



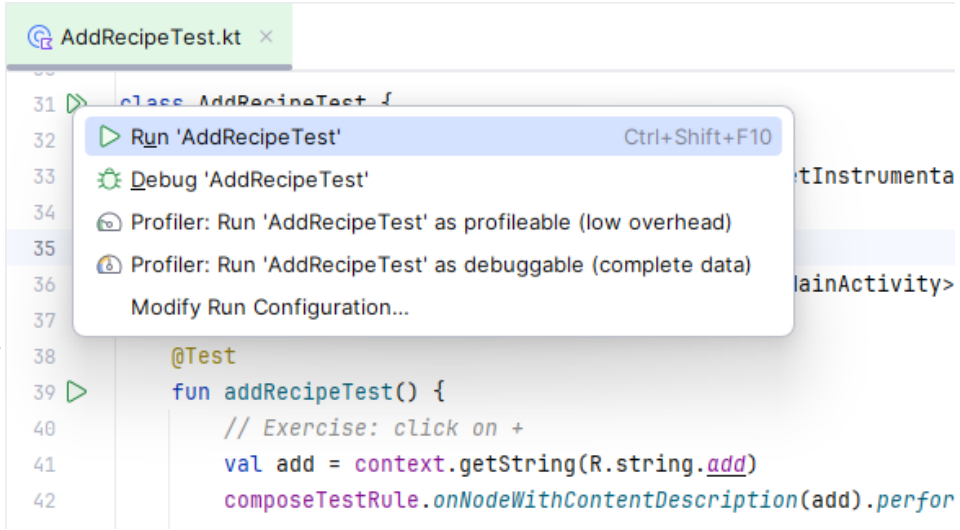
4. Automated tests in Android - Android tests


```
class AddRecipeTest {

    private val context = InstrumentationRegistry.getInstrumentation().targetContext

    @get:Rule
    val testRule = createAndroidComposeRule<MainActivity>()

    @Test
    fun addRecipeTest() {
        // Exercise: click on +
        val add = context.getString(R.string.add)
        testRule.onNodeWithContentDescription(add).performClick()

        // Exercise: add recipe
        testRule.onNodeWithText(context.getString(R.string.name))
            .performTextInput("My recipe")
        testRule.onNodeWithText(context.getString(R.string.ingredients))
            .performTextInput("My ingredients")
        val accept = context.getString(R.string.accept)
        testRule.onNodeWithText(accept).performClick()

        // Verify: we're back to home
        testRule.onNodeWithContentDescription(accept).assertIsNotDisplayed()
        testRule.onNodeWithContentDescription(add).assertIsDisplayed()
    }

}
```

`@Test` annotation marks this as a test function that will be executed by the testing framework (i.e., JUnit)

This test uses our SUT's UI in the same way that a final user would (exercise) and the verifies the UI is as expected

We can execute the test in Android Studio using this button



4. Automated tests in Android - Android tests


```
class FetchTodosTest {

    private val context = InstrumentationRegistry.getInstrumentation().targetContext

    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @Test
    fun fetchTodosTest() {
        // Exercise: click on get todos button
        composeTestRule.onNodeWithText(context.getString(R.string.get_todos)).performClick()

        // Verify: assess resulting todos list
        composeTestRule.waitUntil(5000) {
            composeTestRule.onNodeWithText(context.getString(R.string.my_todos)).isDisplayed()
        }
        val todoItems = composeTestRule.onNodeWithTag("todos").onChildren().fetchSemanticsNode
        assertTrue(todoItems.isNotEmpty())
    }

}
```

This test verifies other part of the UI, waiting until the results are available in the screen as expected

4. Automated tests in Android - Unit tests

- Unit testing in Android apps can be challenging since we need to isolated the unit under test
  - A potential *unit* can be those classes used to implement REST clients (strictly, this would be integration test)

```
interface DummyJsonService {

    @GET("todos")
    suspend fun getTodos(): Response<Todos>

    @POST("recipes/add")
    suspend fun addRecipe(@Body recipe: Recipe): Response<Recipe>

}
```

The problem is that these clients are implemented with coroutines

build.gradle.kts (app)

```
testImplementation(libs.kotlinx.coroutines.test)
```

libs.version.toml

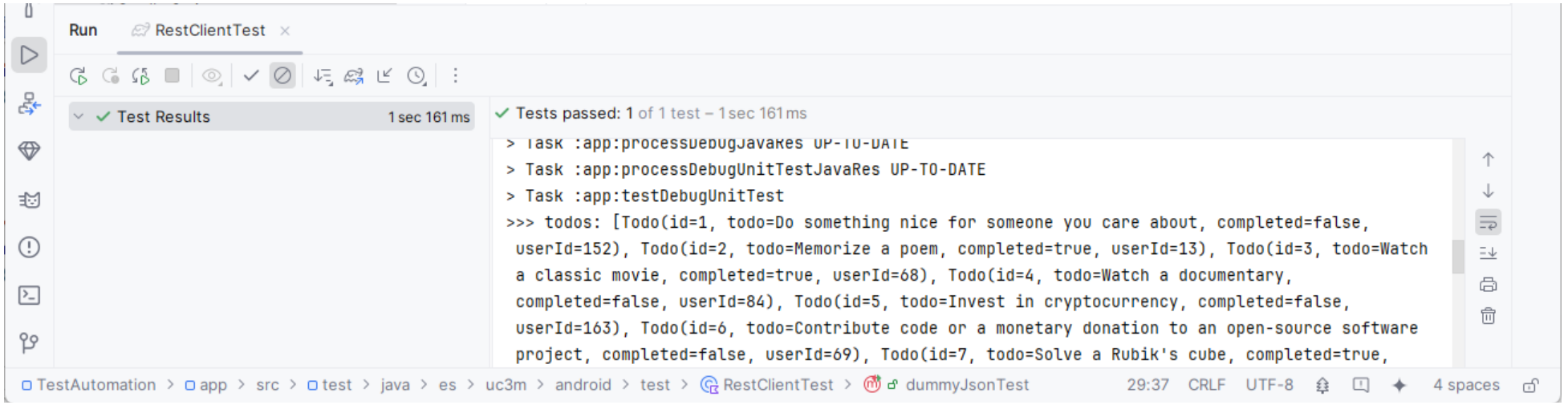
```
[versions]
kotlinxCoroutinesTest = "1.10.2"

[libraries]
kotlinx-coroutines-test = { module = "org.jetbrains.kotlinx:kotlinx-coroutines-test", version.ref = "kotlinxCoroutinesTest" }
```

The solution is to use an specific library for testing coroutines

4. Automated tests in Android - Unit tests

```
class RestClientTest {

    @Test
    fun dummyJsonTest() = runTest {
        // Exercise
        val response = DummyJsonClient.apiService.getTodos()

        // Verify
        assertTrue(response.isSuccessful)
        var todos = response.body()?.todos!!
        println(">>> todos: $todos")
        assertTrue(todos.isNotEmpty())
    }
}
```

runTest is a coroutine test runner that handles asynchronous code execution



4. Automated tests in Android - Unit tests

- Another potential *unit* can be the view model:

```
class RestViewModel : ViewModel() {
    private val _todos = MutableStateFlow<List<Todo>>(emptyList())
    val todos: StateFlow<List<Todo>> = _todos.asStateFlow()

    private val _isLoading = MutableStateFlow(false)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    private val _snackMessage = MutableStateFlow<String?>(null)
    val snackMessage: StateFlow<String?> = _snackMessage.asStateFlow()

    fun fetchTodos() {
        viewModelScope.launch {
            _isLoading.value = true
            try {
                val response = DummyJsonClient.apiService.getTodos()
                if (response.isSuccessful) {
                    _todos.value = response.body()?.todos ?: emptyList()
                }
            } catch (e: Exception) {
                _snackMessage.value = e.message
            } finally {
                _isLoading.value = false
            }
        }
    }


    fun setSnackMessage(message: String?) {
        _snackMessage.value = message
    }

}
```

The problem is that these view models uses a coroutine scope provided by Android (`viewModelScope`). But we want to execute a unit test, not an Android test

4. Automated tests in Android - Unit tests

```
@OptIn(ExperimentalCoroutinesApi::class)
class ViewModelTest {
    private val testDispatcher = UnconfinedTestDispatcher()

    @Before
    fun setup() {
        Dispatchers.setMain(testDispatcher)
    }

    @Test
    fun viewModelTest() = runTest {
        val viewModel = RestViewModel()
        viewModel.fetchTodos()

        val await = Awaitility.await().atMost(Duration.ofSeconds(5))
        await.until { viewModel.todos.value.isNotEmpty() }

        val todos = viewModel.todos.value
        println("*** todos: $todos")
    }

    @After
    fun tearDown() {
        Dispatchers.resetMain()
    }

}
```

A solution is to use a coroutine test dispatcher (i.e., a thread to be used instead of the thread pool provided in the coroutine Android scope)

We replace the main dispatcher because unit tests do not run in Android environment

To do that, in addition to the test runner (*runTest*), we need some mechanism to wait the response, such as Awaitility

build.gradle.kts (app)

```
testImplementation(libs.awaitility)
```

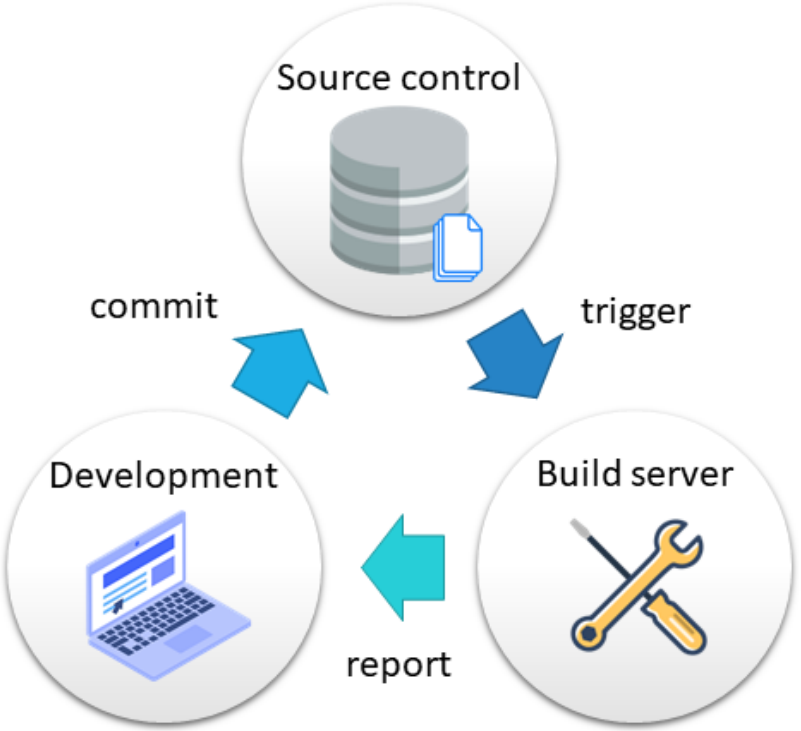
libs.version.toml

```
[versions]
awaitility = "4.3.0"

[libraries]
waitility = { module = "org.awaitility:awaitility", version.ref = "awaitility" }
```

Table of contents

5. Continuous integration

- **Continuous Integration** (CI) is a software development strategy where the members of a software project build, test, and integrate changes (commits) continuously in three stages:



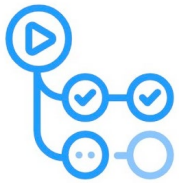
Source code is typically managed using a **version control system** (like Git or CVS)

A **build server**, also known as a CI server, is a dedicated system that automates the process of building, testing, and deploying software applications

CI is sometimes extended to **Continuous Deployment** (CD), in which every change that passes the pipeline is released automatically to production

5. Continuous integration - Build server

- A **build server** is server-side infrastructure that implements CI pipelines (sometimes called *workflows*)
  - A CI pipeline is a series of steps executed to build/test/deploy a given software
- Some popular build servers/platforms are:



GitHub Actions

https://docs.github.com/en/actions

CI/CD

https://docs.gitlab.com/ee/ci/

Jenkins

https://www.jenkins.io/

Bamboo

https://www.atlassian.com/software/bamboo

5. Continuous integration - GitHub Actions

- You can find a separate GitHub repository (different than the usual for examples) with a complete example of app, tests, and CI configuration
- Each time a new commit is done in the repo, the whole test suite (local and instrumented) is executed in GitHub Actions
  - When some test fails (*regression*), the development team is notified

https://github.com/bonigarcia/android-basic-app

Table of contents

6. Takeaways

- Software testing consists of the dynamic evaluation of a piece of software (SUT), giving a verdict about it
- Automated testing improves feedback speed, repeatability, and long-term efficiency
- In Android, we commonly distinguish between local tests (unit) and Android tests (E2E)
- Android Studio supports testing with JUnit and Jetpack Compose testing libraries
- In professional development, teams use CI/CD pipelines to automatically build, test, and sometimes deploy their apps