Tradefed and its relation

> What is tradefed in android explain in simple ways and it's relation with cts vts

In simple terms:

💡 What is Tradefed in Android?

Tradefed (short for **Trade Federation**) is a **test framework** developed by Google to run automated tests on Android devices. It's used to **run, manage, and report** tests like CTS, VTS, and other Android platform-level tests.

Think of Tradefed as a **test manager** or **test engine** that:

- Connects to devices
- Loads test cases
- Executes them
- Collects results
- Logs everything

📦 Relation to CTS and VTS

Component	Purpose	Uses Tradefed?
CTS (Compatibility Test Suite)	Tests **Android APIs** to ensure apps run correctly	✅ Yes
VTS (Vendor Test Suite)	Tests **low-level components** (kernel, HAL, drivers)	✅ Yes

So:

- **CTS** tests **framework-level compatibility** (e.g., `ActivityManager`, `Camera`, etc.)
- **VTS** tests **vendor-level compatibility** (e.g., HALs, kernel, binder, etc.)
- **Both CTS and VTS use Tradefed to run the tests**.

🧱 How does it work?

1. You run a command like:

```
bash

./run_cts
```

2. This launches the **Tradefed test harness**.
3. Tradefed:
    - Detects attached Android devices
    - Loads test modules
    - Executes them in order
    - Collects pass/fail logs
    - Outputs results in XML/HTML

🗂️ Example Tools Using Tradefed

- **`run_cts`** → starts CTS using Tradefed
- **`run_vts`** → starts VTS using Tradefed
- **`vts-tradefed`** → a Tradefed instance pre-configured for VTS
- **`cts-tradefed`** → same but for CTS

🏁 Summary

- **Tradefed** is a **test runner framework**.
- It's the **core engine behind CTS and VTS**.
- It handles **device detection, test execution, and reporting**.
- You don't usually call Tradefed directly — you use wrapper scripts like **`run_cts`**.

Let me know if you want a visual flow or a command walkthrough!

> Who wrote cts and vts

📌 Who wrote CTS and VTS?

Both the **CTS (Compatibility Test Suite)** and **VTS (Vendor Test Suite)** were developed and are maintained by **Google**, specifically by the **Android Open Source Project (AOSP)** team.

👥 Which team exactly?

- ✅ **CTS** is developed by the **Android Compatibility team** at Google.
  - Purpose: To ensure that **Android apps** work consistently across all devices.
  - Tests: Java framework APIs, behavior contracts, permissions, intents, etc.
- ✅ **VTS** is developed by the **Android Systems Engineering and Vendor Interface teams**.
  - Purpose: To ensure that **Android vendors (OEMs, SoC manufacturers)** meet the required **HAL**, **kernel**, and **low-level API** standards.
  - Tests: HALs (via HIDL/AIDL), Linux kernel interfaces, syscalls, SELinux, etc.

🏗️ Where is the source code?

All the source code is in the **AOSP** (Android Open Source Project):

- CTS:
  - AOSP repo: `cts/`
- VTS:
  - AOSP repo: `test/vts/`
- Tradefed:
  - AOSP repo: `tools/tradefederation/`

🛠️ Contributions

- Mostly written by Google engineers.
- Some contributions from OEMs (Samsung, Qualcomm, etc.) and community partners for test coverage.
- Tests evolve with each Android version — new CTS/VTS tests are added to match new APIs or HALs.

🧠 Summary

Component	Developed By	Purpose
CTS	Android Compatibility Team (Google)	Tests app compatibility and Android APIs
VTS	Android Vendor Interface & Systems Team (Google)	Tests vendor compliance with Android HAL/kernel requirements

Let me know if you'd like to see a sample CTS or VTS test source code too.

> It's depends on android version to version?

Yes, **CTS and VTS absolutely depend on the Android version.**

✅ Why do CTS and VTS depend on the Android version?

Each Android version introduces:

- New APIs
- New HALs
- Behavior changes
- Security updates
- Kernel or system interface updates

To **verify compliance**, the CTS and VTS must be **updated** to test those **new or changed features**.

🔁 What changes from version to version?

Area	Changes With Android Version
CTS	- New API tests for new SDK features - Deprecated/removed APIs - Behavior changes (e.g., permissions, background restrictions)
VTS	- New HAL interfaces (via HIDL or AIDL) - New kernel requirements - Updated SEPolicy rules - ABI stability enforcement
Tradefed	- May be updated to support new test formats or modules

---

📦 Example

Android 13 → Android 14:

- CTS added tests for:
  - New APIs in SDK 34
  - New permission behaviors
- VTS added:
  - Tests for new AIDL HALs
  - Kernel interface restrictions (via KMI)

- More strict SELinux policy enforcement

🔪 Important Notes

- **Each Android version has its own CTS/VTS release**.
- Device makers **must pass that version's CTS/VTS** to be certified for that Android version.
- Google publishes test results and tools here:
  - CTS: source.android.com/compatibility/cts
  - VTS: source.android.com/compatibility/vts

🏁 Summary

✅ Yes, **CTS and VTS are version-dependent**.
📅 New Android version → New CTS/VTS → New requirements.

Would you like to see how a device passes CTS for Android 14 vs Android 15?

> Test runner
> Explain this in simple ways and what it's role

Sure! Let's break down the term **Test Runner** in the simplest way:

✅ What is a Test Runner?

A **Test Runner** is like a **test manager** or **robot** that:

- **Finds** test cases
- **Runs** them one by one
- **Records** the results (pass/fail)
- **Reports** what happened

🏃 Real-world analogy:

Imagine you're a school teacher with a stack of student exam papers.
You're the **Test Runner** if you:

1. Open each paper

2. Grade each question
3. Note whether the student passed or failed
4. Write a final report card

🛠️ In Android, what does a Test Runner do?

In Android testing, a **Test Runner** is a **software component** that:

- Loads test classes (like unit tests, CTS/VTS modules)
- Executes them on a **device or emulator**
- Gathers results and logs

📌 Examples of Android Test Runners

Test Runner	Used For	Notes
JUnit Runner	Local Java unit tests	Runs Java test code
AndroidJUnitRunner	Instrumentation tests on device	Runs tests that interact with UI and Android APIs
Tradefed (Trade Federation)	CTS, VTS, platform tests	Advanced test runner built by Google for system-level testing

---

🎯 Role of a Test Runner

Role	Description
🧠 Discover	Finds all test methods (e.g. `testCameraWorks()`)
🏃 Execute	Runs them on the target device/emulator
📊 Report	Shows which tests passed/failed and why
🔗 Connect	Manages communication with device, logs, test configs

---

🔪 Example

You run:

```
bash

./run_cts --module CameraTest
```

Here's what happens:

1. Tradefed (Test Runner) starts

2. Finds `CameraTest` module
3. Connects to your Android device
4. Executes all tests in that module
5. Saves logs and shows results

✅ Summary

- A **Test Runner** is a **tool that runs tests** and **reports results**.
- In Android, test runners are used to **automate and validate app or system behavior**.
- Tools like **Tradefed**, **JUnit**, or **AndroidJUnitRunner** are all examples of test runners.

Let me know if you'd like a diagram or code example!

Trade Federation Overview

Trade Federation overview

bookmark_border
Trade Federation (Tradefed or TF for short) is a continuous test framework designed for running tests on Android devices. For example, Tradefed is used to run the Compatibility Test Suite (CTS) and Vendor Test Suite (VTS).

Trade Federation is a Java application which runs on a host computer, and communicates to one or more Android devices using ddmlib (the library behind DDMS) over adb.

We've listed some of TF's main features below, along with a couple sample usecases. That said, if you want to jump right in and get started, you can head straight for the Start Here page.

Features
modular, flexible, scalable design
has built in support for running many different types of Android tests: instrumentation, uiautomator, native/gtest, host-based JUnit, etc
provides reliability and recovery mechanisms on top of adb
supports scheduling and running tests on multiple devices in parallel
See Testing Through TF for the most up-to-date information about how to run your existing tests, such as Instrumentation.

Use cases
Trade Federation's modularity makes it straightforward to slot into environments with existing build, test, and reporting infrastructures. We list below a few demonstrative usecases where tradefed could enable efficient, scalable test practices.

First, it is useful to consider the landscape of potential usecases in terms of the question "which parts are modifiable, and what parts are static?" For instance, a Device OEM can modify the framework, the system, and the hardware, but has little or no influence over existing applications. An application developer, on the other hand, can modify the app, but has little control over most aspects of the system or the framework.

As a result, an entity in each usecase will have different testing goals, and will have different options in the case of a set of test failures. Despite these differences, Trade Federation can help make each of their test processes efficient, flexible, and scalable.

Device OEM
A Device OEM builds hardware, and will often tweak the Android system and frameworks to run

well on that hardware. The OEM might strive to accomplish those goals while retaining stability and performance at the hardware and system levels, and making sure the framework changes don't break compatibility with existing applications.

The OEM could implement a device flashing module that will execute during the Target Setup stage of the lifecycle. That module would have complete control over the device during its execution period, which would allow it to potentially force the device into the bootloader, flash, and then force the device to reboot back into userspace mode. Combined with a module to tie into a continuous build system, this would allow the OEM to run tests on their device as they make changes to the system-level firmware and Java-level frameworks.

Once the device is fully booted, the OEM would be able to leverage existing JUnit-based tests, or write new ones, to verify the functionality of interest. Finally, they could write one or more result reporting modules to tie into existing test-result repositories, or to report results directly (for instance, by email).

App developer
An Application Developer builds an app which needs to run well across a variety of platform versions and a variety of devices. If an issue comes up on a particular platform version and/or device, the only remedy is to add a workaround and move on. For larger developers, the test process might be incorporated into a continuous build sequence. For smaller developers, it might be kicked off periodically or by hand.

Most app developers would use the apk test installation modules that already exist in TF. There's a version that installs from the local filesystem, as well as a version that can install apks downloaded from a build service. It is important to note that the latter version would continue to work properly with arbitrarily many TF instances running on the same host machine.

Because of TF's proficiency at dealing with multiple devices, it would be straightforward to classify each test result by the type of device that was used for that test. Thus, TF could potentially generate a 2-dimensional (or multi-dimensional) compatibility matrix for every build of the application.

Testing service
A Test Service might, for instance, allow app developers to submit apps and run tests on devices instrumented with power-measurement tools to determine power usage for the app. This differs from the prior two usecases in that the service builder does not control the devices or the applications that are being run.

Because Trade Federation can run any Java class that implements the simple IRemoteTest interface, it's trivial to write drivers that can coordinate some external piece of hardware with the test case that's being run on the device. The driver itself can spawn Threads, send requests to other servers, or do anything else that it might need. Moreover, the simplicity and versatility of the result reporting interface, ITestInvocationListener, means that it is likewise straightforward to represent arbitrary test results (including, for instance, numerical power metrics) into the standard result reporting pipeline.

Here's a **clear and concise overview of Trade Federation (Tradefed or TF)** in Android:

🚀 What is Trade Federation?

Trade Federation (TF) is a **continuous testing framework** used primarily for **automated testing on Android devices**. It is:

- A **Java-based application** that runs on a **host machine**.
- Communicates with one or more Android devices over **ADB** using **ddmlib** (the same library used by DDMS).
- Commonly used to run:
  - **CTS (Compatibility Test Suite)**
  - **VTS (Vendor Test Suite)**

🛠️ Key Features of Tradefed

- ✅ **Modular, flexible, scalable** architecture.
- ✅ Supports various test types:
  - Instrumentation tests
  - UIAutomator tests
  - Native (GTest) tests
  - Host-based JUnit tests
- ✅ Handles **recovery and reliability** over ADB.
- ✅ Can **schedule and execute tests in parallel across multiple devices**.
- ✅ Plug-and-play for use in **existing CI/CD systems**.

🎯 Use Cases

1. Device OEMs

- Modify Android **framework, system, and hardware**.
- Use Tradefed to:
  - **Flash devices** via bootloader in the setup phase.
  - **Run tests automatically** after flashing.
  - Integrate with CI to test system firmware and framework changes.

- Report results via custom modules (email, dashboards, etc.).

2. Application Developers

- Modify only the **app**, not the system or framework.
- Use Tradefed to:
  - **Install APKs** (locally or from remote build systems).
  - Test across **multiple devices and Android versions**.
  - Generate a **compatibility matrix** for app builds.
  - Fit testing into **automated or manual CI pipelines**.

3. Test Services

- Offer **device farms** to test unknown apps or external devices.
- Tradefed enables:
  - Custom test drivers using the `IRemoteTest` interface.
  - Integration with **external hardware (e.g., power measurement tools)**.
  - Custom result reporters via `ITestInvocationListener`.
  - Reporting of custom metrics (e.g., battery drain) using standard formats.

🧩 Key Interfaces

- `IRemoteTest` – Interface for defining custom test logic.
- `ITestInvocationListener` – Interface to report and record test results.

📌 Summary

Aspect	Description
Language	Java
Runs On	Host machine
Talks to Devices	Via ADB (`ddmlib`)
Test Types	Instrumentation, UIAutomator, Native, JUnit
Use Cases	OEM testing, App CI testing, External test services
Strengths	Modularity, parallelism, extensibility, recovery

Would you like a step-by-step tutorial or example to run your first TF test?

Get started with TF

bookmark_border
Trade Federation is a large test infrastructure that can be adapted to a great variety of different usecases, and most people will probably only need some subset of its functionality. First off, we envision that TF users will fulfill any of three primary roles: Developer, Integrator, and Test Runner. A particular individual might wear any (or all) of those three hats, but we feel the distinction will help make it easier to navigate the documentation.

Developers
Developers spend the majority of their time creating TF modules that are written in Java. They may write configurations and execute tests, but will typically only do so to verify that their modules are being invoked properly and are functioning as expected.

Integrators
Integrators spend the majority of their time creating XML test configurations, or command files (which are written in a simple shell-like language). They tie together TF modules written by the Developer with specific configurations that are required for particular test requirements and goals.

Test Runners
Test Runners spend the majority of their time executing tests and generally making sure that test results are being generated, and that the generated test results are relevant, reproducible, and accurate. They spend the majority of their time interacting with tradefed's command line interface, and will also verify that the results make sense.

In order to get the most out of Trade Federation, all three roles will need to be represented. It will take Developers and Integrators to make TF interoperate with other pieces of infrastructure, like build systems and test result repositories. It will take Integrators and Test Runners to get TF to actually run the desired tests and produce the desired test results. It will take Test Runners to identify results that don't make sense, and to work with the Developers and Integrators to figure out where the bugs may lie and get them fixed.

What's next
People in all three roles should at least glance through all the rest of the docs. Machine Setup will get you to the point where you can run TF (by building or downloading it). Working with Devices will explain how to run tests with a physical device, with an emulator, or with no device at all. The Test Lifecycle page will explain from a theoretical perspective how the roles of the Developer, Integrator, and Test Runner interact, and then Option Handling will demonstrate how to put that theory into practice.

Finally, the End-to-End Example takes you through the development, integration, and deployment of a sample test. It involves aspects of each role, and should offer hints at how to

do more complicated things that aren't directly discussed in the documentation.

If you've gotten through everything here and still have unanswered questions, first try taking a look at the Trade Federation source code. Beyond that, feel free to try asking on the android-platform Google Group. For best results, make sure to mention "Trade Federation" (or "tradefed", or "TF") in the message subject.

Here's a **beginner-friendly guide to getting started with Trade Federation (TF)**, organized by roles and next steps:

🎭 Roles in Trade Federation

Understanding your role helps you know what parts of TF you'll interact with the most.

👩‍💻 1. Developer

- **Main task**: Write **Java modules** for custom test logic, device setup, or result reporting.
- Example activities:
  - Implement `IRemoteTest` to define a new test.
  - Create custom `ITestDevice`, `ITargetPreparer`, or `ITestInvocationListener`.

🧩 2. Integrator

- **Main task**: Connect modules using **XML configs** or **command files**.
- Example activities:
  - Create `.xml` test configurations that combine modules.
  - Define test plans to run specific tests under certain conditions.

🚀 3. Test Runner

- **Main task**: Use **command-line interface (CLI)** to run tests and review results.
- Example activities:
  - Run commands like `run cts` or `run vts`.
  - Monitor logs, analyze failures, ensure consistency in test results.

✅ **Real-world overlap**: Most people will play **more than one role**, especially when working in small teams.

🧰 Machine Setup (Next Steps)

To use TF, you'll need to:

1. Build or Download Tradefed

- **Option A: Build from AOSP**

```
bash

source build/envsetup.sh
lunch aosp_arm64-eng
make tradefed-all
```

- Binaries will be in: **`out/host/linux-x86/framework/tradefed.jar`**
- **Option B: Download prebuilt**
  - Available from Android CTS/VTS release pages.

---

2. Run TF

Use the Tradefed console:

```
bash

java -jar tradefed.jar
```

You'll get a TF shell like:

```
text

Welcome to the TF console!
Type 'help' for more information.
```

Example to run CTS:

```
bash

run cts --plan CTS
```

🔌 Working with Devices

TF supports:

- 📱 **Physical devices** (via ADB)
- 🧪 **Emulators**
- 🚫 **No device** (for host-only tests or mocks)

Commands:

```
bash

list devices
run commandAndExit test-command.xml
```

🔄 Test Lifecycle (Conceptual Flow)

Phase	Role(s) Involved	Purpose
Module Development	Developer	Write Java test or utility modules
Integration	Integrator	Configure tests via XML or command files
Execution & Analysis	Test Runner	Run tests and analyze output

⚙️ Option Handling

TF modules often support command-line or XML options:

```
xml

<option name="test-file" value="path/to/test.apk"/>
```

Command line override:

```
bash

run cts --test-file=myapp.apk
```

📚 Learn by Example

✅ End-to-End Example

- Walks you through:
  - Writing a basic **`IRemoteTest`**
  - Creating the XML config
  - Running the test
  - Collecting and viewing results

🔍 This example shows how **Developer**, **Integrator**, and **Test Runner** roles combine in practice.

📌 Summary

Role	What You Do	Main Tools
Developer	Write test logic in Java	**`IRemoteTest`**, **`ITestInvocationListener`**
Integrator	Connect modules with configs	XML, command files
Test Runner	Execute and monitor test runs	TF console, CLI

💬 Still stuck?

- Check the **Tradefed source code** in AOSP:

```
bash

tools/tradefederation/
```

- Ask on android-platform Google Group — include the word "tradefed" in the subject.

Would you like a full working example test (Java + config + run command) to try it locally?

↪
🎭 Roles in Trade Federation ---------- Understanding your role helps you know what parts of TF you'll interact with the most. ### 👩‍💻 1. Developer * **Main task**: Write **Java modules** for custom test logic, device setup, or result reporting. * Example activities: * Implement `IRemoteTest` to define a new test. * Create custom `ITestDevice`, `ITargetPreparer`, or `ITestInvocationListener`. ### 🧩 2. Integrator * **Main task**: Connect modules using **XML configs** or **command files**. * Example activities: * Create `.xml` test configurations that combine modules. * Define test plans to run specific tests under certain conditions. ### 🚀 3. Test Runner * **Main task**: Use **command-line interface (CLI)** to run tests and review results. * Example activities: * Run commands like `run cts` or `run vts`. * Monitor logs, analyze failures, ensure consistency in test results.
explain in simple words

Sure! Here's a **simple explanation** of the three main roles in Trade Federation (TF):

👩‍💻 1. Developer — *The Coder*

- **What they do:**
  Write Java code that tells TF **how to test something**.
- **Example:**
  They might write a test that checks if a camera works or if an app crashes.
- **Think of them as:**
  The people who create new types of tests or custom logic for testing.

🧩 2. Integrator — *The Connector*

- **What they do:**
  Take the test code from developers and **set it up to run** using **XML configuration files**.
- **Example:**
  They decide **which tests should run**, on **what devices**, and **in what order**.
- **Think of them as:**
  The ones who assemble all the pieces to make the test run correctly.

🚀 3. Test Runner — *The Operator*

- **What they do:**
  Run the tests using the command line and **check the results**.
- **Example:**
  They type commands like `run cts` to start testing, then read the output to see if anything failed.
- **Think of them as:**
  The people who press "start", watch the test, and read the results to catch bugs.

🧠 Summary (In One Line):

- **Developer:** Builds the tests.
- **Integrator:** Sets up how the tests will run.
- **Test Runner:** Runs the tests and checks the results.

Would you like a simple diagram or real-world example (like testing a phone app)?

↪

make tradefed-all

> what is this command

The command:

```
css

make tradefed-all
```

is used to **build the Trade Federation (TF) testing framework** from **AOSP (Android Open Source Project)** source code.

🔍 Simple Explanation:

- **`make`**: This is the build system command used in AOSP. It compiles code.
- **`tradefed-all`**: This is a specific target that tells the build system to **build everything needed to run Trade Federation**.

📦 What it builds:

It compiles and creates the **`tradefed.jar`** file (the main file you need to run TF) and other related tools.

After this build, you can find the output here:

```
bash

out/host/linux-x86/framework/tradefed.jar
```

You can then run Tradefed like this:

```
bash

java -jar out/host/linux-x86/framework/tradefed.jar
```

✅ When to use:

Run **`make tradefed-all` after setting up and syncing your AOSP source code**, especially if you plan to:

- Run CTS/VTS or custom TF-based tests.
- Develop or modify Tradefed modules yourself.

Would you like a step-by-step on how to build and run TF from AOSP?

TF development environment

bookmark_border
Trade Federation is distributed with the AOSP and uses the Android build system to create its binary. Make sure you've established a build environment to compile and run packages from the Android source tree.

Build Tradefed
From the root directory of the Android source tree:

```
source ./build/make/envsetup.sh
lunch <device-target>
make tradefed-all -j8
```

See Developing Tradefed for more details about how to set up your environment for coding in Tradefed.

Run from command line
Tradefed requires the adb utility in your $PATH:

```
export PATH=$PATH:<path/to/adb>
```

If Tradefed was built, the tradefed.sh launcher script is accessible from your path. To launch the Trade Federation Console:

```
tradefed.sh
```

Now your environment is set up for Trade Federation.

Here's a **simple step-by-step guide** to set up the **Trade Federation (TF) development environment**:

🛠️ 1. Get the AOSP Source

Trade Federation is part of AOSP. You must have the full Android source code downloaded.

🧱 2. Set Up the Build Environment

From the root of your AOSP source tree, run:

```
bash

source ./build/make/envsetup.sh
```

This sets up the environment for building Android components like TF.

🍽️ 3. Choose a Device Target

Now select the build configuration for your device or emulator:

```
bash

lunch <device-target>
```

Example:

```
bash

lunch aosp_x86_64-eng
```

🔨 4. Build Trade Federation

Run this command to build TF and its tools:

```
bash

make tradefed-all -j8
```

✅ This creates **`tradefed.jar`** in:

```
bash

out/host/linux-x86/framework/tradefed.jar
```

🚀 5. Add `adb` to Your Path

TF needs **`adb`** (Android Debug Bridge) in your **`PATH`** to talk to devices. If it's not already there:

```
bash

export PATH=$PATH:<path/to/adb>
```

Example:

```
bash

export PATH=$PATH:$ANDROID_BUILD_TOP/out/host/linux-x86/bin
```

💻 6. Launch Tradefed Console

If everything was built correctly, run:

```
bash

tradefed.sh
```

This opens the **TF command-line interface**:

```
pgsql

Welcome to the TF console!
Type 'help' for more information.
```

✅ Now You're Ready!

You can now:

- Run tests (like CTS, VTS)
- Develop your own test modules in Java
- Use the TF shell to list devices, schedule tests, and collect results

---

Would you like help writing and running your first custom TF test?

> Work with devices in TF
>
> bookmark_border
> Trade Federation uses an abstraction called ITestDevice to run tests. This abstraction objectifies the lowest-common-denominator Android device:
>
> It has a serial number
> It has a state: Online, Available, Recovery, or Not Available
> It has some notion of reliability. For instance, if we run a command, we can differentiate between the case where the command hasn't finished yet, the case where the device doesn't support running commands, and the case where the device has become unresponsive while running the command.

Device classes
The three primary implementations of ITestDevice represent three common usecases.

Physical device
This is an actual piece of hardware, connected to the TF host machine either by USB, or by using adb's TCP feature. The TestDevice class sits atop the ddmlib library, which is a Java interface to adb. So any physical device listed in adb devices can be instantiated and used as a TestDevice.

Emulator
Emulators are handled specially by TF because they live in another process. To interact with an Emulator, specify the --emulator argument for the command. See LocalSdkBuildProvider and SdkAvdPreparer for more info.

No device
Suppose you have a test that doesn't interact with a device at all. For instance, it might just download a file from some service and verify that the file itself is valid. The NullDevice is an ITestDevice that is just a stub. It has a serial number like null-device-N, and most attempted operations either no-op silently or throw.

Here's a **simple explanation** of how Trade Federation (TF) works with devices:

📱 How TF Handles Devices: `ITestDevice`

In TF, a device (real or virtual) is represented by an object called **`ITestDevice`**.
This object helps TF talk to devices in a **standardized way**, no matter what type they are.

✅ Every `ITestDevice` Has:

- A **serial number** (like from **`adb devices`**)
- A **state**, such as:
  - **`Online`**: connected and ready
  - **`Available`**: ready to run tests
  - **`Recovery`**: TF is trying to fix or reboot it
  - **`Not Available`**: disconnected or unresponsive
- A **way to check reliability**, like:
  - Is the command still running?
  - Did the device go offline?
  - Does it even support shell commands?

🔌 Types of Devices in TF

TF supports three main types of devices using `ITestDevice`:

---

1. 🧱 Physical Device

- **Real hardware** plugged in via USB or over Wi-Fi (ADB TCP).
- Uses the `TestDevice` class in TF.
- Shows up when you run:

```
bash

adb devices
```

Example use case: Testing on a Pixel phone connected to your computer.

---

2. 🖥️ Emulator

- Android **emulator** running on your computer.
- Needs special handling because it runs in its own process.
- Use the `--emulator` flag to let TF know:

```
bash

run my-test --emulator
```

- TF uses helper modules like:
  - `LocalSdkBuildProvider`: downloads the SDK
  - `SdkAvdPreparer`: sets up and launches an AVD

Example use case: Automated tests on virtual Android devices.

---

3. 🚫 No Device (`NullDevice`)

- A **dummy device** used when **no Android device is needed**.
- Example:
  - A test that checks a downloaded file
  - A host-side test that verifies an API

Serial number looks like: `null-device-1`

📌 Summary

Device Type	Class Used	When to Use
Physical device	**`TestDevice`**	Testing real phones/tablets
Emulator	With **`--emulator`**	Testing in Android emulator
No device	**`NullDevice`**	Host-only tests, no device needed

Would you like a working example showing how to run a test on each of these device types using TF?

Trade Federation Console

bookmark_border
The Trade Federation Console is an optional component that allows you to inspect the state of Trade Federation and what it sees, from the device states to the current tests in progress.

It is a great tool for monitoring a large number of tests running in parallel and understanding the progress of each test.

Reach the console
Once Tradefed is built, the tradefed.sh launcher script is accessible from your path and by default will take you to the console.

The console presents itself with the tf > prompt.

What can the console do?
The help of the console will always list the most up-to-date information. A few interesting features of it are:

List the devices and how Tradefed sees their state: list devices
List the currently running invocations and their metadata: list invocations
Get the logs of all running invocations and Tradefed: dump logs
The console allows you to debug what is happening in Tradefed and the devices by querying some states that are not shown together anywhere else.

How do I disable the console?
Sometimes the console isn't needed, such as when running a one-time command. The console must be disabled when running Tradefed from a script or when piping its output directly. Running Tradefed from a script with the console enabled can cause unexpected behavior.

To prevent the console from starting, run the tradefed.sh launcher script with the commandAndExit argument.

tradefed.sh run commandAndExit <usual command>
Console autocompletion
The console provides basic autocompletion of configuration names.

tf > run <hit TAB>
result in:
Display all 167 possibilities? (y or n)
By hitting TAB after the run you can get the full list of configurations available. And if you have a partial name already typed in, the console will print all the possibilities.

tf >run tf/<HIT TAB>

tf/acceptance tf/fake tf/func
tf/stress tf/uiautomator tf/unit-runner
tf >run tf/
This is a useful when you don't remember an exact configuration name.

Here's a **simple explanation** of the **Trade Federation Console** in Android:

💻 What is the Tradefed Console?

The **Trade Federation Console** is a **command-line tool** that opens when you run:

```
bash

tradefed.sh
```

It shows a prompt like:

```
nginx

tf >
```

This console helps you:

- See what devices are connected
- Check which tests are running
- View logs and test progress

- Run and debug test commands easily

📋 What Can You Do in the Console?

Here are some **useful commands** you can run inside the console:

Command	What It Does
`list devices`	Shows connected devices and their status
`list invocations`	Lists all running tests
`dump logs`	Shows logs of tests and Tradefed itself
`run <test>`	Starts a test
`help`	Lists all available console commands

🚫 How to Disable the Console

If you're running TF in a **script or automated environment**, the interactive console can cause issues.

Use this instead:

```
bash

tradefed.sh run commandAndExit <your-command>
```

Example:

```
bash

tradefed.sh run commandAndExit cts --plan CTS
```

✅ This runs the command and exits directly — no prompt or console.

⚡ Console Autocompletion

You can press **`TAB`** to autocomplete test names.

Example:

```
bash

tf > run <TAB>
```

This will show you all available tests you can run.

If you type part of a name:

```
bash

tf > run tf/ui<TAB>
```

It will show matching options like:

```
bash

tf/uiautomator
```

🎯 Summary

Feature	Description
tf > prompt	Appears when you run **`tradefed.sh`**
Interactive commands	Check devices, tests, logs, etc.
Disable with script	Use **`commandAndExit`**
Autocompletion	Press **`TAB`** to explore test configs

Would you like an example test to try inside the console (e.g., a basic CTS run)?

TF test lifecycle

bookmark_border
The lifecycle of a test executed using Trade Federation is composed of four separate stages, designed around formally defined interfaces.

Defined interfaces
Build Provider: Provides a build to test, downloading appropriate files if necessary.
Target Preparer: Prepares the test environment, possibly including software installation and device configuration.
Test: Executes test(s) and gathers test results. This may be any JUnit Test, although our IRemoteTest interface is specifically designed to work well in the Trade Federation environment.
Test Invocation Listener (results reporting): Listens for test results, usually for the purpose of forwarding the test results to a repository or displaying them to the Test Runner.
The fundamental testing entity in TF is a Configuration (config). A config is an XML file that declares the lifecycle components of a test.

This separation of the test's lifecycle is intended to allow for reuse. Using this design, the Developer can create a Test once, and then the Integrator can create different Configurations to

run that Test in different environments. For example, they could create a Configuration that will run a test on a local machine and dump the result to stdout. They could then create a second Configuration that would execute that same test, but use a different Test Invocation Listener to store the test results in a database. A third Configuration might be designed run that test continuously from a test lab somewhere.

It's convenient to note here that a Configuration along with its command-line arguments (as provided by the Test Runner) is known as a Command. When TF pairs a Command with an ITestDevice and executes it, the subsequent object is known as an Invocation. In short, an Invocation encompasses a complete TF test execution, across its entire lifecycle.

Additional configuration components
Device Recovery: mechanism to recover device communication if lost.
Logger: collects tradefed logging data.
Stage output and errors
Each stage of an invocation executes sequentially and has a specific goal. This section describes the usual outputs and errors of each stage.

Build provider
This stage creates and outputs an IBuildInfo object that contains all the required files references to set up and run the tests.

The most common error at this stage is a failure to download or find the requested files.

An error at this stage results in directly reporting the error, and no tests being run.

Target preparation
This stage sets up the necessary states for the target under tests. This stage can alter the device or the host setup as needed for the given test invocation.

Common errors at this stage usually involve failure to setup the device into a given state (for example, failed flashing) and failure to find the required files for the setup.

An error at this stage results in target clean up running, reporting of the error, and no tests being run.

Tests
This stage runs the requested tests on the previously prepared target, and reports all the test execution results.

Common errors at this stage usually involve the target under test being unavailable or some error causing partial execution of the tests. These errors are infrastructure issues that affect the test execution itself as opposed to a failure of a single test case.

An error at this stage results in the test execution stopping, the target clean up running, reporting the error, and getting partial results.

> Results reporting
> This stage reports the results and errors to the configured services (for example, servers and local files).
>
> Although individual results reporters can have errors, they're isolated from each other (one reporter doesn't see errors from another one). These errors affect only an individual reporter's own results reporting and the errors can be viewed in the logs.

Here's a **simple explanation** of the **Trade Federation (TF) test lifecycle**, broken down step by step:

🔄 Trade Federation Test Lifecycle: 4 Stages

Every test run in Trade Federation goes through **4 main stages**, and each stage is handled by a **well-defined interface**:

🏗️ 1. Build Provider

> "Get the files needed to run the test"

- **What it does**: Downloads or provides the **software build**, app, or other files needed for the test.
- **Creates**: An object called `IBuildInfo` with all required file paths.
- **Common errors**: Missing files, download failure.
- **If it fails**: **Test doesn't run at all**.

⚙️ 2. Target Preparer

> "Prepare the device or test environment"

- **What it does**: Sets up the device or host for the test.
  - Examples: Flash the device, install an APK, set system properties.
- **Errors**: Failed flashing, bad config, missing setup files.
- **If it fails**: TF **cleans up**, reports error, and **skips the test**.

🔪 3. Test

> "Run the actual test"

- **What it does**: Runs the test cases on the device or host.

  - Can be: JUnit test, Instrumentation test, GTest, etc.
  - Must implement `IRemoteTest`.
- **Errors**: Device goes offline, crash during test, partial results.
- **If it fails**: TF stops, does cleanup, and **returns partial results**.

📝 4. Result Reporting

"Send the test results somewhere"

- **What it does**: Collects and reports test results.
  - Could be to: files, dashboards, email, or a results server.
  - Handled by: `ITestInvocationListener`.
- **Errors**: Only affect that specific reporter; results are still saved by others.

🧩 Extra Components

Component	Purpose
Device Recovery	Tries to fix devices if they go offline
Logger	Logs messages and errors during the run

📦 Configuration = XML + Modules

- Each test run is described by a **Configuration file** (an XML file).
- It lists:
  - Which `BuildProvider`, `TargetPreparer`, `Test`, and `Listener` to use.
- You can **reuse the same test** in different ways by changing the config:
  - Dump results to console
  - Save them to a file
  - Upload to a server

⚙️ Command and Invocation

Term	Meaning
Command	A config file + CLI arguments
Invocation	A full test run on a specific device

✅ One **Invocation** = full end-to-end test process:
Build → Prepare → Run Test → Report

🧠 Summary Table

Stage	Purpose	Common Failures	Outcome if Failed
Build Provider	Get build & test files	Download fail, missing files	Test doesn't run
Target Preparer	Setup device or host	Flash fail, config errors	Test skipped + error
Test	Run the actual test	Device crash, ADB loss	Partial results + cleanup
Result Reporter	Report results	One reporter fails (others continue)	Logs show the error

---

Would you like a diagram or real-world example to help visualize this flow?

Option handling in Tradefed

bookmark_border
Option handling lies at the heart of Trade Federation's modular approach. In particular, options are the mechanism by which the developer, Integrator, and Test Runner can work together without having to duplicate each-other's work. Put simply, our implementation of option handling allows the developer to mark a Java class member as being configurable, at which point the value of that member can be augmented or overridden by the Integrator, and can be subsequently augmented or overridden by the Test Runner. This mechanism works for all Java intrinsic types, as well as for any Map or Collection instances of intrinsic types.

Note: The option-handling mechanism only works for classes implementing one of the interfaces included in the Test Lifecycle, and only when that class is instantiated by the lifecycle machinery.

Developer
To start off, the developer marks a member with the @Option annotation. They specify (at a minimum) the name and description values, which specify the argument name associated with that Option, and the description that is displayed on the TF console when the command is run with --help or --help-all.

As an example, let's say we want to build a functional phone test that dials a variety of phone numbers, and expects to receive a sequence of DTMF tones from each number after it connects.

```
public class PhoneCallFuncTest extends IRemoteTest {
   @Option(name = "timeout", description = "How long to wait for connection, in millis")
```

```
    private long mWaitTime = 30 * 1000;  // 30 seconds

    @Option(name = "call", description = "Key: Phone number to attempt. " +
            "Value: DTMF to expect. May be repeated.")
    private Map<String, String> mCalls = new HashMap<String, String>;

    public PhoneCallFuncTest() {
        mCalls.add("123-456-7890", "01134");  // default
    }
```

That's all that's required for the developer to set up two points of configuration for that test. They could then go off and use mWaitTime and mCalls as normal, without paying much attention to the fact that they're configurable. Because the @Option fields are set after the class is instantiated, but before the run method is called, that provides an easy way for implementors to set up defaults for or perform some kind of filtering on Map and Collection fields, which are otherwise append-only.

Integrator

The Integrator works in the world of configurations, which are written in XML. The config format allows the Integrator to set (or append) a value for any @Option field. For instance, suppose the Integrator wanted to define a lower-latency test that calls the default number, as well as a long-running test that calls a variety of numbers. They could create a pair of configurations that might look like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration description="low-latency default test; low-latency.xml">
    <test class="com.example.PhoneCallFuncTest">
        <option name="timeout" value="5000" />
    </test>
</configuration>

<?xml version="1.0" encoding="utf-8"?>
<configuration description="call a bunch of numbers; many-numbers.xml">
    <test class="com.example.PhoneCallFuncTest">
        <option name="call" key="111-111-1111" value="#*#*TEST1*#*#" />
        <option name="call" key="222-222-2222" value="#*#*TEST2*#*#" />
        <!-- ... -->
    </test>
</configuration>
```

Test Runner

The Test Runner also has access to these configuration points through the Trade Federation console. First and foremost, they run a command (that is, a config and all of its arguments) with the run command <name> instruction (or run <name> for short). Beyond that, they can specify any list of arguments are part of the command, which can replace or append to fields specified by lifecycle objects within each config.

To run the low-latency test with the many-numbers phone numbers, the Test Runner could execute:

tf> run low-latency.xml --call 111-111-1111 #*#*TEST1*#*# --call 222-222-2222 #*#*TEST2*#*#
Or, to get a similar effect from the opposite direction, the Test Runner could reduce the wait time for the many-numbers test:

tf> run many-numbers.xml --timeout 5000
Option ordering
You might notice that the call option underlying implementation is a Map so upon repeated --call on the command line, they're all stored.

The option timeout, which has an underlying implementation of long, can only store one value. So only the last value specified is stored. --timeout 5 --timeout 10 results in timeout containing 10.

In case of a List or Collection as the underlying implementation, all the values are stored, in the order specified on the command line.

Boolean options
Options of boolean underlying type can be set to true by directly passing the option name, for example, --[option-name] and can be set to false using the syntax --no-[option-name].

See also
Pass options to suite and modules

Let's break **Tradefed Option Handling** into **simple terms**.

🔧 What is "Option Handling" in Tradefed?

Option handling lets you **configure Java fields (variables) from outside the code**, like using a command line or an XML file — without changing the source code.

This is useful when:

- **Developers** write a test with configurable values.
- **Integrators** modify these values in XML files.
- **Test runners** (you) can override them using command-line flags.

👩‍💻 Developer's Role

In Java code, the developer uses **`@Option`** to say:

> "This variable should be configurable."

Example:

```
java

@Option(name = "timeout", description = "Time to wait")
private long mWaitTime = 30000;  // default is 30 seconds

@Option(name = "call", description = "Phone number and expected tone")
private Map<String, String> mCalls = new HashMap<>();
```

This means:

- **`timeout`** can be changed.
- **`call`** entries can be added from outside.

🔧 Integrator's Role (XML Config)

Integrator writes a **config XML** like:

```
xml

<test class="com.example.PhoneCallFuncTest">
    <option name="timeout" value="5000" />
</test>
```

This sets the **`timeout`** to 5 seconds.

Another example:

```
xml

<option name="call" key="123" value="DTMF1" />
<option name="call" key="456" value="DTMF2" />
```

This sets phone number–tone pairs.

🚀 Test Runner's Role (Command Line)

Test runners can **override or add values** on the command line:

```
bash

tf> run low-latency.xml --call 111-111-1111 #*#*TEST1*#*#
```

Or override timeout:

```
bash

tf> run many-numbers.xml --timeout 5000
```

🧠 Option Types (How they behave)

- **Single value types** (e.g., `int`, `long`, `String`): only the **last value is used**.

  ```
  bash

  --timeout 5 --timeout 10  → timeout = 10
  ```

- **Map or Collection**: all values are added.

  ```
  r

  --call A X --call B Y  → mCalls = {A: X, B: Y}
  ```

- **Boolean**:
  - `--featureX` → true
  - `--no-featureX` → false

🎯 Summary

Who	What They Do
Developer	Marks variables as `@Option`
Integrator	Sets values in XML config
Test Runner	Changes values with command line args

This system avoids hardcoding and allows easy test customization without code changes.

Let me know if you want a working code + XML + tf command demo.

> yes

Here's a **simple explanation** of **Option handling in Tradefed**:

💡 What is Option Handling?

Option handling in Trade Federation (Tradefed) is a **way to make test parameters configurable** using annotations in Java (**`@Option`**), so that:

- **Developers** can write reusable test classes.
- **Integrators** can configure them via XML files.
- **Test runners** can override or add options via the command line.

👩‍💻 For Developers: Use `@Option` Annotation

Developers use **`@Option`** in their Java test code to make variables configurable.

Example:

```java
@Option(name = "timeout", description = "Wait time in milliseconds")
private long mWaitTime = 30000;  // Default is 30 seconds
```

This means **`mWaitTime`** can now be changed via XML config or command line later.

You can also use Maps and Lists:

```java
@Option(name = "call", description = "Phone number and expected DTMF")
private Map<String, String> mCalls = new HashMap<>();
```

🛠️ For Integrators: Configure in XML

They write XML files to provide values for those **`@Option`**s.

Example:

```xml
<test class="com.example.PhoneCallFuncTest">
    <option name="timeout" value="5000" />
</test>
```

Or for multiple phone calls:

```xml
<test class="com.example.PhoneCallFuncTest">
    <option name="call" key="111-111-1111" value="#*#*TEST1*#*#" />
```

```
    <option name="call" key="222-222-2222" value="#*#*TEST2*#*#" />
</test>
```

🪛 For Test Runners: Pass via Command Line

They can override or add options when running the test:

```
bash

tf> run low-latency.xml --call 111-111-1111 #*#*TEST1*#*# --call 222-222-2222 #*#*TEST2*#*#
```

Or:

```
bash

tf> run many-numbers.xml --timeout 5000
```

⚙️ Option Behavior by Type

Option Type	Behavior
`long/int`	Only **last value** is kept
`List/Collection`	All values are stored **in order**
`Map`	Each key-value pair is stored
`boolean`	**`--option-name`** = true **`--no-option-name`** = false

🧠 Summary

- Use **`@Option`** in Java to make things configurable.
- Use XML to define different test setups.
- Use command line to override/add to configs.
- Helps **reuse tests** with different parameters, easily.

Let me know if you want me to walk through the phone call test example in detail too.

End-to-end TF test example

bookmark_border

This tutorial guides you through creating a "hello world" Trade Federation (Tradefed or TF) test configuration and gives you a hands-on introduction to the TF framework. Starting from a development environment, you will create a simple configuration and add features.

The tutorial presents the test development process as a set of exercises, each consisting of several steps, that demonstrate how to build and gradually refine your configuration. All sample code you need to complete the test configuration is provided, and the title of each exercise is annotated with a letter describing the roles involved in that step:

D for Developer
I for Integrator
R for Test Runner
After completing the tutorial, you will have a functioning TF configuration and understand many important concepts in the TF framework.

Set up Trade Federation
For details on setting up the TF development environment, see Machine Setup. The rest of this tutorial assumes you have a shell open that has been initialized to the TF environment.

For simplicity, this tutorial illustrates adding a configuration and its classes to the TF framework core library. This can be extended to developing modules outside the source tree by compiling the tradefed JAR, then compiling your modules against that JAR.

Create a test class (D)
Lets create a hello world test that just dumps a message to stdout. A tradefed test generally implements the IRemoteTest interface. Here's an implementation for the HelloWorldTest:

```
package com.android.tradefed.example;

import com.android.tradefed.device.DeviceNotAvailableException;
import com.android.tradefed.invoker.TestInformation;
import com.android.tradefed.log.LogUtil.CLog;
import com.android.tradefed.result.ITestInvocationListener;
import com.android.tradefed.testtype.IRemoteTest;

public class HelloWorldTest implements IRemoteTest {
   @Override
   public void run(TestInformation testInfo, ITestInvocationListener listener) throws
DeviceNotAvailableException {
       CLog.i("Hello, TF World!");
   }
}
```

Save this sample code to
<tree>/tools/tradefederation/core/src/com/android/tradefed/example/HelloWorldTest.java and rebuild tradefed from your shell:

m -jN
Note that CLog.i in the example above is used to direct output to the console. More information on logging in Trade Federation is described in Logging (D, I, R).

If the build does not succeed, consult Machine Setup to ensure you didn't miss a step.

Create a configuration (I)
Trade Federation tests are made executable by creating a Configuration, an XML file that instructs tradefed on which test (or tests) to run, as well as which other modules to execute and in what order.

Lets create a new Configuration for our HelloWorldTest (note the full class name of the HelloWorldTest):

```
<configuration description="Runs the hello world test">
    <test class="com.android.tradefed.example.HelloWorldTest" />
</configuration>
```

Save this data to a helloworld.xml file anywhere on your local filesystem (e.g. /tmp/helloworld.xml). TF will parse the Configuration XML file (aka config), load the specified class using reflection, instantiate it, cast it to a IRemoteTest, and call its run method.

Run the config (R)
From your shell, launch the tradefed console:

```
tradefed.sh
```

Ensure a device is connected to the host machine and is visible to tradefed:

```
tf> list devices
Serial          State     Product   Variant   Build   Battery
004ad9880810a548  Available  mako      mako      JDQ39   100
```

Configurations can be executed using the run <config> console command. Try:

```
tf> run /tmp/helloworld.xml
05-12 13:19:36 I/TestInvocation: Starting invocation for target stub on build 0 on device
004ad9880810a548
Hello, TF World!
```

You should see "Hello, TF World!" output on the terminal.

You can confirm that a command is done running by using list invocations or l i in the console prompt, and it should print nothing. If commands are currently running, they display as follows:

```
tf >l i
Command Id  Exec Time  Device       State
10          0m:00      [876X00GNG]  running stub on build(s) 'BuildInfo{bid=0, target=stub, serial=876X00GNG}'
```

Add the config to the classpath (D, I, R)
For convenience of deployment, you can also bundle configs into the tradefed JARs themselves. Tradefed automatically recognizes all configurations placed in config folders on the classpath.

To illustrate, move the helloworld.xml file into the tradefed core library (<tree>/tools/tradefederation/core/res/config/example/helloworld.xml). Rebuild tradefed, restart the tradefed console, then ask tradefed to display the list of configurations from the classpath:

```
tf> list configs
[...]
example/helloworld: Runs the hello world test
```

You can now run the helloworld config using:

```
tf> run example/helloworld
05-12 13:21:21 I/TestInvocation: Starting invocation for target stub on build 0 on device 004ad9880810a548
Hello, TF World!
```

Interact with a device (D, R)
So far, our HelloWorldTest isn't doing anything interesting. Tradefed's specialty is running tests using Android devices, so lets add an Android device to the test.

Tests can get a reference to an Android device by using TestInformation, provided by the framework when the IRemoteTest#run method is called.

Let's modify the HelloWorldTest print message to display the serial number of the device:

```
@Override
public void run(TestInformation testInfo, ITestInvocationListener listener) throws DeviceNotAvailableException {
    CLog.i("Hello, TF World! I have device " + testInfo.getDevice().getSerialNumber());
}
```

Now rebuild tradefed and check the list of devices:

```
tradefed.sh
```

```
tf> list devices
Serial          State     Product   Variant   Build   Battery
004ad9880810a548  Available  mako      mako      JDQ39   100
```

Take note of the serial number listed as Available; that is the device that should be allocated to HelloWorld:

```
tf> run example/helloworld
05-12 13:26:18 I/TestInvocation: Starting invocation for target stub on build 0 on device
004ad9880810a548
Hello, TF World! I have device 004ad9880810a548
```

You should see the new print message displaying the serial number of the device.

Send test results (D)

IRemoteTest reports results by calling methods on the ITestInvocationListener instance provided to the #run method. The TF framework itself is responsible for reporting the start (via ITestInvocationListener#invocationStarted) and end (via ITestInvocationListener#invocationEnded) of each Invocation.

A test run is a logical collection of tests. To report test results, IRemoteTest is responsible for reporting the start of a test run, the start and end of each test, and the end of the test run.

Here's what the HelloWorldTest implementation might look like with a single failed test result.

```
@Override
public void run(TestInformation testInfo, ITestInvocationListener listener) throws
DeviceNotAvailableException {
    CLog.i("Hello, TF World! I have device " + testInfo.getDevice().getSerialNumber());

    TestDescription testId = new TestDescription("com.example.TestClassName", "sampleTest");
    listener.testRunStarted("helloworldrun", 1);
    listener.testStarted(testId);
    listener.testFailed(testId, "oh noes, test failed");
    listener.testEnded(testId, Collections.emptyMap());
    listener.testRunEnded(0, Collections.emptyMap());
}
```

TF includes several IRemoteTest implementations you can reuse instead of writing your own from scratch. For example, InstrumentationTest can run an Android application's tests remotely on an Android device, parse the results, and forward those results to the ITestInvocationListener). For details, see Test Types.

Store test results (I)

The default test listener implementation for a TF config is TextResultReporter, which dumps the results of an invocation to stdout. To illustrate, run the HelloWorldTest config from the previous section:

./tradefed.sh

tf> run example/helloworld
04-29 18:25:55 I/TestInvocation: Invocation was started with cmd: /tmp/helloworld.xml
04-29 18:25:55 I/TestInvocation: Starting invocation for 'stub' with '[BuildInfo{bid=0,
target=stub, serial=876X00GNG} on device '876X00GNG']
04-29 18:25:55 I/HelloWorldTest: Hello, TF World! I have device 876X00GNG
04-29 18:25:55 I/InvocationToJUnitResultForwarder: Running helloworldrun: 1 tests
04-29 18:25:55 W/InvocationToJUnitResultForwarder:
Test com.example.TestClassName#sampleTest failed with stack:
oh noes, test failed
04-29 18:25:55 I/InvocationToJUnitResultForwarder: Run ended in 0 ms
To store the results of an invocation elsewhere, such as in a file, specify a custom ITestInvocationListener implementation using the result_reporter tag in your configuration.

TF also includes the XmlResultReporter listener, which writes test results to an XML file in a format similar to that used by the ant JUnit XML writer. To specify the result_reporter in the configuration, edit the .../res/config/example/helloworld.xml config:

```
<configuration description="Runs the hello world test">
    <test class="com.android.tradefed.example.HelloWorldTest" />
    <result_reporter class="com.android.tradefed.result.XmlResultReporter" />
</configuration>
```

Now rebuild tradefed and re-run the hello world sample:

tf> run example/helloworld
05-16 21:07:07 I/TestInvocation: Starting invocation for target stub on build 0 on device 004ad9880810a548
Hello, TF World! I have device 004ad9880810a548
05-16 21:07:07 I/XmlResultReporter: Saved device_logcat log to /tmp/0/inv_2991649128735283633/device_logcat_6999997036887173857.txt
05-16 21:07:07 I/XmlResultReporter: Saved host_log log to /tmp/0/inv_2991649128735283633/host_log_6307746032218561704.txt
05-16 21:07:07 I/XmlResultReporter: XML test result file generated at /tmp/0/inv_2991649128735283633/test_result_536358148261684076.xml. Total tests 1, Failed 1, Error 0
Notice the log message stating that an XML file has been generated; the generated file should look like this:

```
<?xml version='1.0' encoding='UTF-8' ?>
<testsuite name="stub" tests="1" failures="1" errors="0" time="9" timestamp="2011-05-
```

```
17T04:07:07" hostname="localhost">
  <properties />
  <testcase name="sampleTest" classname="com.example.TestClassName" time="0">
    <failure>oh noes, test failed
    </failure>
  </testcase>
</testsuite>
```

You can also write your own custom invocation listeners—they simply need to implement the ITestInvocationListener interface.

Tradefed supports multiple invocation listeners, so you can send test results to multiple independent destinations. To do this, just specify multiple <result_reporter> tags in your config.

Logging facilities (D, I, R)
TF's logging facilities include the ability to:

Capture logs from the device (aka device logcat)
Record logs from the Trade Federation framework running on the host machine (aka host log)
The TF framework automatically captures the logcat from the allocated device and sends it to the invocation listener for processing. XmlResultReporter then saves the captured device logcat as a file.

TF host logs are reported using the CLog wrapper for the ddmlib Log class. Let's convert the previous System.out.println call in HelloWorldTest to a CLog call:

```
@Override
public void run(ITestInvocationListener listener) throws DeviceNotAvailableException {
    CLog.i("Hello, TF World! I have device %s", getDevice().getSerialNumber());
```

CLog handles string interpolation directly, similar to String.format. When you rebuild and rerun TF, you should see the log message on stdout:

```
tf> run example/helloworld
...
05-16 21:30:46 I/HelloWorldTest: Hello, TF World! I have device 004ad9880810a548
...
```

By default, tradefed outputs host log messages to stdout. TF also includes a log implementation that writes messages to a file: FileLogger. To add file logging, add a logger tag to the config, specifying the full class name of FileLogger:

```
<configuration description="Runs the hello world test">
    <test class="com.android.tradefed.example.HelloWorldTest" />
    <result_reporter class="com.android.tradefed.result.XmlResultReporter" />
    <logger class="com.android.tradefed.log.FileLogger" />
```

</configuration>
Now, rebuild and run the helloworld example again:

```
tf >run example/helloworld
...
05-16 21:38:21 I/XmlResultReporter: Saved device_logcat log to
/tmp/0/inv_6390011618174565918/device_logcat_1302097394309452308.txt
05-16 21:38:21 I/XmlResultReporter: Saved host_log log to
/tmp/0/inv_6390011618174565918/host_log_4255420317120216614.txt
...
```

The log message indicates the path of the host log, which, when viewed, should contain your HelloWorldTest log message:

```
more /tmp/0/inv_6390011618174565918/host_log_4255420317120216614.txt
```

Example output:

```
...
05-16 21:38:21 I/HelloWorldTest: Hello, TF World! I have device 004ad9880810a548
```

Handling options (D, I, R)

Objects loaded from a TF Configuration (aka Configuration objects) can also receive data from command line arguments through the use of the @Option annotation.

To participate, a Configuration object class applies the @Option annotation to a member field and provides it a unique name. This enables that member field value to be populated via a command line option (and also automatically adds that option to the configuration help system).

Note: Not all field types are supported. For a description of supported types, see OptionSetter.

Let's add an @Option to HelloWorldTest:

```
@Option(name="my_option",
        shortName='m',
        description="this is the option's help text",
        // always display this option in the default help text
        importance=Importance.ALWAYS)
private String mMyOption = "thisisthedefault";
```

Next, let's add a log message to display the value of the option in HelloWorldTest so we can demonstrate it was received correctly:

```
@Override
public void run(ITestInvocationListener listener) throws DeviceNotAvailableException {
   ...
   CLog.logAndDisplay(LogLevel.INFO, "I received option '%s'", mMyOption);
```

Finally, rebuild TF and run helloworld; you should see a log message with the my_option default value:

```
tf> run example/helloworld
...
05-24 18:30:05 I/HelloWorldTest: I received option 'thisisthedefault'
```

Pass values from the command line

Pass in a value for my_option; you should see my_option populated with that value:

```
tf> run example/helloworld --my_option foo
...
05-24 18:33:44 I/HelloWorldTest: I received option 'foo'
```

TF configurations also include a help system, which automatically displays help text for @Option fields. Try it now, and you should see the help text for my_option:

```
tf> run example/helloworld --help
Printing help for only the important options. To see help for all options, use the --help-all flag

 cmd_options options:
   --[no-]help         display the help text for the most important/critical options. Default: false.
   --[no-]help-all     display the full help text for all options. Default: false.
   --[no-]loop         keep running continuously. Default: false.

 test options:
   -m, --my_option     this is the option's help text Default: thisisthedefault.

 'file' logger options:
   --log-level-display  the minimum log level to display on stdout. Must be one of verbose,
debug, info, warn, error, assert. Default: error.
```

Note the message about "printing only the important options." To reduce option help clutter, TF uses the Option#importance attribute to determine whether to show a particular @Option field help text when --help is specified. --help-all always shows help for all @Option fields, regardless of importance. For details, see Option.Importance.

Pass values from a configuration

You can also specify an Option value within the config by adding a <option name="" value=""> element. Test it using helloworld.xml:

<test class="com.android.tradefed.example.HelloWorldTest" >
  <option name="my_option" value="fromxml" />
</test>
Re-building and running helloworld should now produce this output:

05-24 20:38:25 I/HelloWorldTest: I received option 'fromxml'
The configuration help should also update to indicate the default value of my_option:

tf> run example/helloworld --help
 test options:
  -m, --my_option this is the option's help text Default: fromxml.
Other configuration objects included in the helloworld config, such as FileLogger, also accept options. The option --log-level-display is interesting because it filters the logs that show up on stdout. Earlier in the tutorial, you may have noticed the "Hello, TF World! I have device ...' log message stopped being displayed on stdout after we switched to using FileLogger. You can increase the verbosity of logging to stdout by passing in the --log-level-display arg.

Try this now, and you should see the 'I have device' log message reappear on stdout, in addition to being logged to a file:

tf> run example/helloworld --log-level-display info
...
05-24 18:53:50 I/HelloWorldTest: Hello, TF World! I have device 004ad9880810a548
That's all, folks!
As a reminder, if you're stuck on something, the Trade Federation source code has a lot of useful information that isn't exposed in the documentation. If all else fails, try asking on the android-platform Google Group, with "Trade Federation" in the message subject.

Here's a **simple explanation** of the full Trade Federation (TF) "Hello World" test tutorial you shared:

🔧 What is Trade Federation (TF)?

Trade Federation is a test framework made by Android for running automated tests **on Android devices**, either real or emulated. It supports flexible test execution, logging, and result reporting.

🚀 Step-by-step: How to Run a Simple "Hello World" Test

1. Set up your environment

- You must have the TF development environment set up.
- Use **`tradefed.sh`** to open the TF shell.

---

2. Write a test class 👩‍💻

Create a simple Java test that prints "Hello, TF World!" to the logs.

```
java

public class HelloWorldTest implements IRemoteTest {
    @Override
    public void run(TestInformation testInfo, ITestInvocationListener listener) {
        CLog.i("Hello, TF World!");
    }
}
```

Save it in the correct folder and **rebuild** TF using **`m -jN`**.

---

3. Create a config file 🗂️

TF needs a config (XML) to know **what test class to run**.

```
xml

<configuration description="Runs the hello world test">
    <test class="com.android.tradefed.example.HelloWorldTest" />
</configuration>
```

Save this as **`helloworld.xml`**.

---

4. Run your test 🏃‍♂️

- Start the TF console: **`tradefed.sh`**
- List devices: **`list devices`**
- Run the test:
  `run /tmp/helloworld.xml`

You should see:
`Hello, TF World!`

---

5. Print device serial number 🔢

Update the test to print the connected device's ID:

```
java

CLog.i("Hello, TF World! I have device " + testInfo.getDevice().getSerialNumber());
```

Rebuild and run again.

---

6. Send a fake test result ❌✅

Pretend a test runs and fails:

```
java

TestDescription testId = new TestDescription("com.example.TestClassName", "sampleTest");
listener.testRunStarted("helloworldrun", 1);
listener.testStarted(testId);
listener.testFailed(testId, "oh noes, test failed");
listener.testEnded(testId, Collections.emptyMap());
listener.testRunEnded(0, Collections.emptyMap());
```

---

7. Save test results as XML 📄

Update config:

```
xml

<result_reporter class="com.android.tradefed.result.XmlResultReporter" />
```

This saves results (including logcat) to `/tmp/...` in XML format.

---

8. Log to a file 🗂️

Add this to the config to also save host logs:

```
xml

<logger class="com.android.tradefed.log.FileLogger" />
```

---

9. Handle command-line options 🛠️

Add a field with `@Option`:

```
java

@Option(name="my_option", shortName='m', description="custom message")
private String mMyOption = "thisisthedefault";
```

Print it in your test:

```
java

CLog.i("I received option '%s'", mMyOption);
```

Now run with:

```
sh

run example/helloworld --my_option hello
```

---

10. Pass options from XML 📥

Add inside `<test>` tag:

```
xml

<option name="my_option" value="fromxml" />
```

---

11. Show help ❓

In TF console:

```
sh

run example/helloworld --help
```

Shows all `@Option` fields marked as important.

✅ Summary

You now know how to:

- Create and run a test
- Print logs and device info
- Report fake test results
- Save logs/results to files
- Accept command-line and XML-based options

This is the basic flow of a TF test—from writing a test, configuring it, running it, and collecting results.

Would you like a code template or script to automate any of this?

confusing

No worries — let's **simplify** everything and break it into small, clear pieces.

🌟 What is Trade Federation (TF)?

- It's a tool to **run tests on Android devices**.
- You can test real phones or emulators using TF.
- You write your own tests (in Java), and TF helps run them and collect results.

✅ What are you trying to do?

You're trying to make a **very simple TF test** that says:

> **"Hello, TF World!"**

🔧 Step-by-Step — Super Simple Version

👩‍💻 Step 1: Write this tiny Java test

Create a file like **`HelloWorldTest.java`** with:

```
java


import com.android.tradefed.testtype.IRemoteTest;
import com.android.tradefed.invoker.TestInformation;
import com.android.tradefed.result.ITestInvocationListener;
import com.android.tradefed.log.LogUtil.CLog;

public class HelloWorldTest implements IRemoteTest {
    @Override
    public void run(TestInformation testInfo, ITestInvocationListener listener) {
        CLog.i("Hello, TF World!");
    }
}
```

📌 This is the test. It just prints one line.

⚙️ Step 2: Create a config XML

Make a file **`helloworld.xml`**:

```xml
<configuration description="My Hello World test">
    <test class="com.android.tradefed.example.HelloWorldTest" />
</configuration>
```

📌 This tells TF what class to run.

🏃 Step 3: Run it using TF shell

1. Open TF shell:

```
tradefed.sh
```

2. Run the test:

```pgsql
run /path/to/helloworld.xml
```

3. See the output:

```
Hello, TF World!
```

✅ That's it! You made your first TF test.

Would you like help setting this up step-by-step in your own machine? I can guide you through writing, building, and running it.

> what types of tests we can test here

Great question!

In **Trade Federation (TF)**, you can run many types of tests on Android devices. It's not limited to just one kind. Here's a breakdown in **simple terms**:

🔪 Types of Tests You Can Run in Tradefed

1. JUnit-based Unit Tests

- ✅ Test small pieces of Java code (like a function).
- 🧠 Good for logic correctness.
- 📦 Usually written as `.jar` or `.apk` files.

Example:
Test whether `LoginValidator.isValid(email)` returns `true` for a valid email.

---

2. Device Instrumentation Tests

- 🔧 These are **Android app tests** that run **on the device**.
- 🔪 You can test your Activity, Services, UI, etc.
- 🎯 Based on AndroidJUnitRunner.

Command:

```
bash

run instrument -p com.example.myapp
```

Use case:
Test whether your `MainActivity` launches correctly on the device.

---

3. GTests (Google Tests)

- 🧊 Native C/C++ tests.
- 🛠️ Used to test native code in AOSP like HALs or native libraries.

Command:

```
bash

run gtest --module-name my_native_module
```

Use case:
Testing a native audio driver's behavior.

---

4. Host-side Tests

- 🖥️ These run on your PC (host) and interact with the device using adb.
- ✅ Used to check log files, push/pull files, issue shell commands, etc.

Use case:
Check if a specific file exists on `/data/` after the app runs.

---

5. CTS/VTS Tests

- 📦 **CTS**: Compatibility Test Suite – to ensure Android app and device compatibility.
- ⚙️ **VTS**: Vendor Test Suite – to test HALs, HIDL/AIDL interfaces, kernel, etc.

Command:

```
bash


run cts --module CameraITS
run vts --plan vts
```

---

6. Custom Tests

- You can write **your own test classes** using `IRemoteTest`, like your HelloWorld example.
- Useful for infrastructure testing, automation scripts, device state verification, etc.

🧰 Common Tradefed Commands

Type	Command Example
Unit Test (jar)	`run commandAndJar`
Instrumentation	`run instrument -p <pkg>`
Native GTest	`run gtest --module-name <module>`
CTS	`run cts --module <module>`
VTS	`run vts --plan <plan>`
Custom XML config	`run /path/to/test_config.xml`

Want a Practical Demo?

I can help you:

- Write a **real instrumented test** or
- Create a **GTest for native code** or
- Run a **CTS module** on your phone/emulator

Just tell me what you want to try first!

↳
IRemoteTest
what is this and why we need this

Great! Let's break it down:

🔍 What is `IRemoteTest` in Trade Federation?

`IRemoteTest` is a **Java interface** used in Trade Federation (TF) to define any **test that runs on a device**.

In **simple words**:
➡️ **`IRemoteTest`** is like a **template or contract** for writing your **own custom test logic** in Tradefed.

🧱 Why do we need `IRemoteTest`?

✅ 1. Custom Test Logic

- You can't always use standard tests (like JUnit or instrumentation).
- **`IRemoteTest`** lets you create **custom behaviors**, like:
  - Rebooting the device and verifying a log file.
  - Sending shell commands and checking output.
  - Verifying battery stats, logs, thermal states, etc.

✅ 2. Flexibility

- It gives you full control over:
  - What to run
  - How to run it
  - What to report as pass/fail

✅ 3. Integrates with Tradefed Infra

- You get logging, device control, reporting, result collection, and retry logic **for free**.

🔧 What does it look like?

Here's a **very simple example** of a custom `IRemoteTest`:

```
java

public class HelloWorldTest implements IRemoteTest {
    @Override
    public void run(TestInformation testInfo, ITestInvocationListener listener)
            throws DeviceNotAvailableException {

        listener.testRunStarted("HelloWorld", 1);
        listener.testStarted(new TestDescription("HelloWorldTest", "testPrint"));

        System.out.println("Hello, Tradefed!");

        listener.testEnded(new TestDescription("HelloWorldTest", "testPrint"), new HashMap<>());
        listener.testRunEnded(100, new HashMap<>());
    }
}
```

➡️ This test just **prints "Hello, Tradefed!"** and reports it as a test run.

🔁 What happens when you run it?

1. Tradefed finds your `IRemoteTest` class from the XML config or command.
2. It runs your `run()` method.
3. You control exactly what happens inside — shell commands, logic, validation, etc.
4. Tradefed reports the results back.

📦 Example Use Cases

- Test **after OTA update**, check version
- Validate **device-specific features** (e.g., custom sensor, custom logging)
- Check presence of certain files or system settings
- Automate **power measurement**, **camera dumps**, or **network status**

TL;DR:

Question	Answer
What is it?	A Java interface to write custom device tests in TF.
Why use it?	For advanced, non-standard test flows on devices.
When to use?	When built-in tests (JUnit, GTest, etc.) aren't enough.
Who uses it?	Platform engineers, OEMs, test teams, automation devs.

Would you like a real working project using `IRemoteTest` with AOSP build instructions?

> Write and run Tradefed tests
>
> bookmark_border
> This section is aimed at developers, or test writers, providing guidance on test execution, as well as writing tests. The instructions are split into two broad categories of tests:
>
> Tests executing directly through Tradefed.
> Tests executing in the context of a suite, for example the Android Compatibility Test Suite (CTS).
> We detail some features that are applicable to both suite and non-suite tests, for example:
>
> Automatic Retries for tests.
> Retry Isolation for tests.
> We also provide some generic end-to-end examples about how to run some of the more common test types, such as:
>
> Execute the instrumentation tests from an existing APK.
> The end-to-end examples above do not require a local checkout of Android. You can simply download Tradefed and use them directly.

Let's break this down **simply** — this is about **how to write and run tests using Tradefed**, either standalone or as part of larger test suites like **CTS (Compatibility Test Suite)**.

🪛 What kinds of tests can you run with Tradefed?

There are **two major categories**:

1. ✅ Standalone Tradefed Tests

- You run them **directly using Tradefed**, usually from command line.
- Good for **custom tests**, development, or device bring-up.
- Examples:
  - Instrumentation tests (from an APK)
  - Shell command tests
  - Java tests implementing `IRemoteTest`

2. 📦 Suite-Based Tests (e.g., CTS, VTS, STS)

- These are part of official **Android test suites**.

- You run them using `run cts` / `run vts` etc.
- Tradefed is the engine under the hood.
- Follow stricter formats for reporting and test structure.

🚀 Types of Common Tests You Can Run

Test Type	Description
Instrumentation	Runs tests in an APK using AndroidJUnitRunner. Example: UI tests, unit tests.
GTest (native)	Runs C++ native tests built into the system or NDK modules.
Host-side JUnit Tests	Java tests that run on the **host (PC)** and control the device via ADB.
ShellCommandTest	Runs and validates output of **`adb shell`** commands.
Custom IRemoteTest	Java test class with custom logic using **`ITestDevice`** APIs.

🔁 Advanced Features

Tradefed provides smart features to improve reliability and debug-ability:

Feature	Description
Automatic Retries	If tests fail, Tradefed can retry them automatically.
Retry Isolation	Only retry the **specific failed tests**, not the entire module.
Isolated Shards	Run tests in **parallel** on multiple devices or threads.
Test Filters	Include or exclude specific test classes or methods.
Test Timeouts	Set how long a test can take before failing due to timeout.

✅ Examples: Running Tests

🔹 1. Run instrumentation test from APK:

```
bash

./tradefed.sh run instrumentation -n com.example.test/androidx.test.runner.AndroidJUnitRunner
```

🔹 2. Run CTS test (as part of suite):

```
bash

./run cts --module CtsDisplayTestCases
```

🔹 3. Run a host test:

```
bash

./tradefed.sh run host -n com.android.myhosttest.MyTest
```

💡 Good to Know

- You **don't need full AOSP checkout** to run some tests. Just install Tradefed and have an APK or test JAR.
- If you are part of OEM, chip vendor, or developer working on devices, **Tradefed is the standard** test framework to use.
- Tradefed integrates into CI pipelines, bugreport collection, retry logic, and report formats.

🚩 Want to Get Started?

I can give you:

- A working **instrumentation test project**
- A working **IRemoteTest Java example**
- A simple shell-based test config XML

Let me know which one you want to try first!

> Test through Tradefed
>
> bookmark_border
> Execution of tests in Tradefed is conducted by test runners that allow the control of Android devices for necessary testing steps. Follow these sections in order.
>
> 1. Write a new test runner
> Here you will learn the basics that go into writing a new Tradefed test runner. This is particularly useful if you plan to support a brand new type of test, and you need a new runner to be developed from scratch.
>
> See Write a Tradefed test runner for instructions.
>
> 2. Write a shardable test runner
> You may also learn how to make your test runner shardable. A shardable test runner allows the

infrastructure to distribute the full test execution over several devices (collocated or not). This is useful when the corpus of tests is large, and you want to parallelize the execution and speed completion.

See Write a sharded IRemoteTest test runner for steps.

3. Write a host-driven test
Host-driven tests are a common use case where test execution is driven from the host-side and queries the device as needed for the test. This is useful when device operation required by the test affects the device state itself, for example rebooting the device.

The test runner type can be used within Tradefed or when running through a suite.

See Write a host-driven test in Trade Federation for instructions.

4. Report metrics from tests
It's fairly common for a test to report metrics in addition to the execution results. Depending on the test runner, there are several methods to report the metrics.

See Report metrics or data from a Tradefed test for examples.

5. Automated log collection
Some logs are commonly used for debugging issues, for example: Logcat. So Tradefed offers an automated mechanism to collect them easily.

See Automated log on failure collection for use.

6. Automatic test retry
You may enable Tradefed to automatically retry failures or run some tests several times in iterations.

See Automatic test retry for more details.

7. Retry isolation
You may enable Tradefed to attempt to isolate the test execution between retry attempts.

See Retry isolation for more details.

8. Write a shell test
See Write a shell test in Trade Federation.

9. Configure global test filters
See Global filters in Trade Federation.

Tradefed (Trade Federation) is a **test harness** developed by Google to run **automated tests** on Android devices. It's widely used in the Android Open Source Project (AOSP) for **testing device behavior**, system components, and app compatibility.

🔍 What is Tradefed?

Tradefed is a **Java-based framework** that allows developers and test engineers to:

- Run tests across one or more Android devices
- Automate testing tasks like rebooting, flashing, logging, etc.
- Collect test results and logs (like logcat, screenshots)
- Retry tests automatically when they fail

✅ Types of Tests You Can Run with Tradefed

Here are common types of tests that Tradefed supports:

Test Type	What It Tests	Example Use Case
Instrumentation tests	App-level tests using **`AndroidJUnitRunner`**	Test your app's UI or logic
Shell tests	Runs shell commands and checks output	Validate system behavior with shell
Host-driven tests	Java tests run on the host controlling the device	Reboot device, check states, compare logs
CTS tests	Compatibility Test Suite (Android certification)	Check AOSP compliance
GTS tests	Google Test Suite (for GMS devices)	Ensure compliance with Google services
VTS tests	Vendor Test Suite (tests HALs, kernel)	HAL testing, device bring-up
JUnit or TF unit tests	Pure Java or system component logic	Unit testing of system components

---

⚙️ Why Tradefed is Useful

You need Tradefed when:

- You want to **automate Android device testing** at scale
- You want to **test Android platform components** (not just apps)
- You're developing or validating **HALs, framework features, or Android builds**
- You want consistent logging, **automated retries**, and **test parallelism**
- You're contributing to AOSP or customizing Android for a device

---

🧪 Example Tests You Can Write or Run

Scenario	Description
App-level test	Run instrumentation test from APK
Host-side test	Control and validate device from host
New test runner	Custom logic to execute test on device
Shell test	Write simple shell commands to validate states
Sharded test	Split tests across devices for speed
Retry-on-failure	Automatically re-run failed tests
Collect logcat/screenshot on failure	Debug with logs automatically

---

🧭 What You Can Do in Tradefed Console

You can use the **interactive console** (`tf >`) to:

- Run tests: **`run cts --module Camera`**
- List devices: **`list devices`**
- Show test status: **`list invocations`**
- Dump logs: **`dump logs`**
- Auto-complete config: **`run <TAB>`**

Would you like me to show how to **run a test** using Tradefed or write a **custom test**?

Write a sharded IRemoteTest test runner

bookmark_border
When writing a test runner, it's important to think about scalability. Ask yourself, "if my test runner had to run 200K test cases" how long would it take?

Sharding is one of the answers available in Trade Federation. It requires splitting all the tests the runner needs into several chunks that can be parallelized.

This page describes how to make your runner shardable for Tradefed.

Interface to implement
The single most important interface to implement to be considered shardable by TF is IShardableTest, which contains two methods: split(int numShard) and split().

If your sharding is going to depend on the number of shards requested, you should implement split(int numShard). Otherwise, implement split().

When a TF test command is executed with sharding parameters --shard-count and --shard-index, TF iterates through all IRemoteTest to look for ones implementing IShardableTest. If found, it will call split to get a new IRemoteTest object to run a subset of test cases for a specific shard.

What should I know about the split implementation?
You runner may shard upon some conditions only; in that case return null when you did not shard.
Try to split as much as it makes sense: split your runner into unit of execution that makes sense for it. It really depends of your runner. For example: HostTest is sharded at the Class level, each test class is put in a separate shard.
If it makes sense, add some options to control the sharding a little bit. For example: AndroidJUnitTest has a ajur-max-shard to specify the maximum number of shards it could split in, regardless of the number requested.
Detailed example implementation
Here is an example code snippet implementing IShardableTest you can reference. The full code is available at (https://android.googlesource.com/platform/tools/tradefederation/+/refs/heads/android16-release/test_framework/com/android/tradefed/testtype/InstalledInstrumentationsTest.java)

```
/**
 * Runs all instrumentation found on current device.
 */
@OptionClass(alias = "installed-instrumentation")
public class InstalledInstrumentationsTest
        implements IDeviceTest, IResumableTest, IShardableTest {
    ...

    /** {@inheritDoc} */
    @Override
    public Collection<IRemoteTest> split(int shardCountHint) {
        if (shardCountHint > 1) {
            Collection<IRemoteTest> shards = new ArrayList<>(shardCountHint);
            for (int index = 0; index < shardCountHint; index++) {
                shards.add(getTestShard(shardCountHint, index));
            }
            return shards;
        }
        // Nothing to shard
        return null;
    }
```

```
    private IRemoteTest getTestShard(int shardCount, int shardIndex) {
        InstalledInstrumentationsTest shard = new InstalledInstrumentationsTest();
        try {
            OptionCopier.copyOptions(this, shard);
        } catch (ConfigurationException e) {
            CLog.e("failed to copy instrumentation options: %s", e.getMessage());
        }
        shard.mShardIndex = shardIndex;
        shard.mTotalShards = shardCount;
        return shard;
    }
    ...
}
```

This example simply creates a new instance of itself and sets shard parameters to it. However, the splitting logic can be totally different from test to test; and as long as it is deterministic and yields collectively exhaustive subsets, it is okay.

Independence
Shards need to be independent! Two shards created by your implementation of split in your runner should not have dependencies on each other or share resources.

Shards splitting needs to be deterministic! This is also mandatory, given the same conditions, your split method should always return the exact same list of shards in the same order.

NOTE: Since each shard can run on different TF instances, it is critical to ensure the split logic yields subsets that are mutually exclusive and collectively exhaustive in a deterministic manner.

Shard a test locally
To shard a test on a local TF, you can simply add the --shard-count option to the command line.

```
tf >run host --class com.android.tradefed.UnitTests --shard-count 3
```

Then TF will automatically spawn commands for each shard and run them.

```
tf >l i
Command Id  Exec Time  Device         State
3           0m:03      [null-device-2]  running stub on build 0 (shard 1 of 3)
3           0m:03      [null-device-1]  running stub on build 0 (shard 0 of 3)
3           0m:03      [null-device-3]  running stub on build 0 (shard 2 of 3)
```

Test result aggregation
Since TF does not do any test result aggregation for sharded invocations, you need to make sure your reporting service supports it.

Let's break this down in **simple words**.

🔧 What is this about?

It's about writing a **test runner** in **Tradefed** that supports **sharding** — meaning it can **split a big test job into smaller pieces**, so they run **in parallel**, speeding things up.

💡 Why do we need this?

Imagine you have **200,000 tests** to run. If you run them all on one device, it will take **hours or even days**.
But if you **split the tests into chunks** and run each chunk on a **separate device or thread**, you finish much faster.

This process is called **sharding**.

📦 Key Interface: `IShardableTest`

To make your test runner support sharding, you implement an interface called **`IShardableTest`**.

It requires one of these methods:

- **`split(int numShard)`** — splits tests into the given number of shards.
- **`split()`** — auto-decides how to split.

✅ What does `split(...)` do?

It should **return a list of smaller test runners** (or test chunks), each running a portion of the total tests.

> Example: If you call **`split(3)`**, you might get 3 mini test runners:
>
> - Runner 0 → runs test 1–10
> - Runner 1 → runs test 11–20
> - Runner 2 → runs test 21–30

📘 Example:

```
java

@Override
public Collection<IRemoteTest> split(int shardCountHint) {
    if (shardCountHint > 1) {
        Collection<IRemoteTest> shards = new ArrayList<>();
        for (int i = 0; i < shardCountHint; i++) {
            shards.add(getTestShard(shardCountHint, i));
        }
```

```
            return shards;
        }
        return null; // If we don't want to shard
    }
```

This code creates multiple copies of the test, each configured to run only **one portion** of the full test list.

🔁 Important Rules:

- **Each shard must run independently.**
- **No two shards should overlap.**
- **Split logic must be deterministic** – given the same input, always return the same shards in the same order.

🔪 How to run sharded tests

```
bash

tradefed> run host --class com.example.MyTest --shard-count 3
```

This will automatically:

- Create 3 test runners.
- Assign them to 3 devices/threads.
- Run them in parallel.

📊 Result aggregation?

Tradefed **does NOT combine** the results from all shards by default.
So your **reporting tool or script** should be smart enough to gather all results from the shards and present them as a single report.

🧠 Summary

Concept	Simple Meaning
Sharding	Splitting tests to run faster in parallel
IShardableTest	Interface to support splitting in your test runner
split(int)	Method to create test chunks
Independent Shards	Shards must not depend on each other
Aggregation	You must handle combining results manually

Let me know if you want a **real Java code example** for your own test runner.

Write a host-driven test in Trade Federation

bookmark_border
This page describes how to write a JUnit4-style device test driven by the host. This means that the host side of the harness is going to trigger actions against the device.

Note that we consider "host-side" tests and "host-driven" tests to be slightly different:

host-driven test: Is a test running on the host that interacts with one or more devices. The system under test (SUT) is not on the host itself but is being tested from the host.
host-side test: Is a test purely running on the host and testing something only on the host, for example unit tests.
Why create a host-driven test rather than an instrumentation test?
Some tests might require you to affect the device overall state, like issuing a reboot command. In the instrumentation test case, a reboot would kill the instrumentation, the test could not continue, and no results would be available.

Host-driven tests can also drive additional setup steps that require interaction with external devices on which the test depends on.

A host-driven test can handle these use cases and allow for advanced testing of the device with more scenarios. If you are in that situation, writing a host-driven test makes the most sense.

How are host-driven tests written in TF?
Here is a sample:

```
@RunWith(DeviceJUnit4ClassRunner.class)
public class SampleHostJUnit4DeviceTest extends BaseHostJUnit4Test {
   @Before
   public void setUp() throws Exception {
     // Some setup
   }

   @Test
   public void testCheckWeHaveDevice() throws Exception {
      Assert.assertNotNull(getDevice());
   }
}
```

Host-driven tests in Trade Federation are driven by the DeviceJUnit4ClassRunner JUnit4 test runner. The overall structure of the test class is the same as a regular JUnit4 test:

@BeforeClass
@Before
@Test
@After
@AfterClass
Assume, Assert
Extending BaseHostJunit4Test is a way to inherit useful testing utilities API such as:

installPackage: Allows to install an APK on the target device.
installPackageAsUser: Allows to install an APK as a user on the target device.
uninstallPackage: Allows to uninstall an APK.
isPackageInstalled: Check whether a package is installed or not.
hasDeviceFeature: Check whether device supports a feature or not. (pm list features)
runDeviceTests(DeviceTestRunOptions options): Run an instrumentation test against a target device using DeviceTestRunOptions to handle all the possible options.
Also provide access to the Tradefed device object:

getDevice(): Returns a TF device object for manipulating the device.
getBuild(): Returns a build info TF object to get information about the build.
getAbi(): Returns the ABI the test is running against.
Tradefed support: Per-class device preparation and clean up
JUnit4 @BeforeClass and @AfterClass are only applicable to static methods, which makes it impossible to use the #getDevice() handler to do some device-specific, one-time, per-class setup or clean up. To solve this issue, use the Tradefed annotation.

@BeforeClassWithInfo: Runs before @BeforeClass annotations
@AfterClassWithInfo: Runs after @AfterClass annotations

```
  @BeforeClassWithInfo
  public static void beforeClassWithDevice(TestInformation testInfo) {
     assertNotNull(testInfo.getDevice());
     testInfo.properties().put("mytest:test-prop", "test");
  }

  @AfterClassWithInfo
  public static void afterClassWithDevice(TestInformation testInfo) {
     assertNotNull(testInfo.getDevice());
     testInfo.properties().put("mytest:test-prop", "test");
  }
```

TestInformation allows you to use the device and store properties that can be used either in the static or non-static scope. BaseHostJUnit4Test supports getting the TestInformation in a non-static scope via #getTestInformation().

If you are not extending BaseHostJUnit4Test, you can implement ITestInformationReceiver in order to receive the TestInformation object.

How to configure a host-driven test in Tradefed?
In Tradefed XML configuration file, host-driven tests are run through the HostTest runner.

```
<test class="com.android.tradefed.testtype.HostTest" >
  <option name="class" value="android.sample.cts.SampleHostJUnit4DeviceTest" />
</test>
```

Let's break this down simply:

✅ What is a host-driven test in Tradefed (Trade Federation)?

A **host-driven test** is a test **written in Java**, executed **on your computer (host)**, and **controls or checks something on the Android device**.

It's **not** running inside the Android app on the device like an **`instrumentation test`**.
Instead, it's like a remote controller test: the **test runs on the host**, but it **sends commands to the device**, checks results, and continues.

🧠 Why do we need host-driven tests?

Some tests require control that you **can't do inside the device app**:

- Need to **reboot** the phone? You can't do that from inside the app (it will kill the test).
- Need to **install/uninstall apps**, change device settings, or control **multiple devices**?
- Need to **test device behavior externally** (e.g., with USB or network setup)?

In these cases, we use **host-driven tests**.

👨‍💻 How do we write a host-driven test in Tradefed?

You write it like a **JUnit4 test**, but extend Tradefed's helper class.

✅ Example:

```java
@RunWith(DeviceJUnit4ClassRunner.class)
public class SampleHostJUnit4DeviceTest extends BaseHostJUnit4Test {

    @Before
    public void setUp() throws Exception {
        // Do any setup before each test
    }

    @Test
```

```
    public void testCheckWeHaveDevice() throws Exception {
        Assert.assertNotNull(getDevice());  // Check we got a device
    }
}
```

🧪 What can we do inside host-driven tests?

By extending **`BaseHostJUnit4Test`**, you get helper functions:

Method	What it does
`getDevice()`	Gets the test device (so you can run commands, install apps, etc.)
`installPackage(path)`	Installs an APK on the device
`uninstallPackage(name)`	Uninstalls an app
`runDeviceTests(...)`	Runs instrumentation tests on the device
`hasDeviceFeature("feature")`	Checks if device supports a hardware/software feature

⚙️ How to configure the test in Tradefed XML?

You use the **`HostTest`** runner and point it to your test class.

```
xml


<test class="com.android.tradefed.testtype.HostTest">
  <option name="class" value="android.sample.cts.SampleHostJUnit4DeviceTest" />
</test>
```

💡 Advanced: One-time setup using annotations

JUnit **`@BeforeClass`** methods are static (can't access device), so Tradefed adds:

Annotation	Purpose
`@BeforeClassWithInfo`	Run one-time setup with device access
`@AfterClassWithInfo`	Run one-time cleanup with device access

These get a **`TestInformation`** object so you can access the device or store test state.

```
java


@BeforeClassWithInfo
public static void beforeClass(TestInformation info) {
    assertNotNull(info.getDevice());
    info.properties().put("key", "value");
}
```

✅ Summary

Concept	Meaning
Host-driven test	Runs on host, controls device
Used for	Reboots, setup/teardown, multi-device tests
Tradefed base class	**`BaseHostJUnit4Test`**
Run with	**`HostTest`** in XML
JUnit features	Use **`@Before`**, **`@Test`**, **`@After`**, etc.
Advanced setup	Use **`@BeforeClassWithInfo`** for per-class setup with device access

Let me know if you'd like a working example project or XML to try it out.

> yes

Great! Let's now break down what a **host-driven test** in **Trade Federation (Tradefed)** is **in simple words**, and **why and how you write one**.

💡 What is a Host-Driven Test?

A **host-driven test** is:

- A **Java test** that runs **on your computer (host)**.
- It controls or interacts with a **real Android device** connected via USB or emulator.
- It is used to test something **on the device**, but the test logic itself is executed **from the host side**.

🔍 Why do we need Host-Driven Tests?

Use host-driven tests **when you want to do something to the device that instrumentation tests cannot handle**, such as:

- **Rebooting** the device (which would kill instrumentation).
- **Installing/uninstalling apps**.
- Running multiple devices together.
- Doing **device setup**, **log checking**, or **external verification** from host.
- Need more **flexibility** than what is possible in an on-device test.

✅ Host-Driven Test vs Instrumentation Test

Feature	Host-Driven Test	Instrumentation Test
Runs on	Host machine (PC)	Android device
Controls device	Yes	No (it is running inside)
Can reboot device?	✅ Yes	❌ No
Can use adb commands	✅ Yes	❌ Limited
Flexibility	✅ High	➖ Medium
Needs installed APK	Optional	Required

🛠️ How to Write a Host-Driven Test

1. **Use JUnit4 style** with annotations like `@Before`, `@Test`, `@After`.
2. **Extend** `BaseHostJUnit4Test` to access helper methods like:
   - `getDevice()` → to get the connected Android device.
   - `installPackage()` → install APKs.
   - `runDeviceTests()` → run instrumentation tests from host.

✍️ Example Code

```
java

@RunWith(DeviceJUnit4ClassRunner.class)
public class SampleHostJUnit4DeviceTest extends BaseHostJUnit4Test {

    @Before
    public void setUp() throws Exception {
        // Setup before test
    }

    @Test
    public void testCheckWeHaveDevice() throws Exception {
        // Check device is connected
        Assert.assertNotNull(getDevice());
    }
}
```

🧩 Extra Features

- Use `@BeforeClassWithInfo` and `@AfterClassWithInfo` for **one-time setup/cleanup** using `TestInformation`.
- `TestInformation` gives access to device and properties at the class level.

📄 XML to Run This Test in Tradefed

Your XML config might look like this:

```
xml

<test class="com.android.tradefed.testtype.HostTest">
    <option name="class" value="android.sample.cts.SampleHostJUnit4DeviceTest" />
</test>
```

📌 Summary

- **Host-driven tests** are **powerful** when you need to control the device from outside.
- Good for testing reboot, install/uninstall, multi-device scenarios.
- Written in **Java using JUnit4**, run by **Tradefed** from your PC.
- Not limited like instrumentation tests.

Would you like a full working Java + XML example you can try on your setup?

Write a host-side deviceless test in TF

bookmark_border
This page tells you how to write a host-side test that doesn't require a device, such as a test that runs on a Linux GCE instance. (For details about writing a host-driven test that requires a device, refer to Write a Host-driven Test in Trade Federation.)

Host-side test types
You can run several types of host-side tests through Trade Federation (TF).

Native (gtest) tests
Create Native tests (gtests) to test a platform. If the test doesn't require a device, run it on a host; the test will run much faster that way. To configure such tests to run on a test host, use the TF runner HostGTest.

This is a sample TradeFed test configuration:

```
<configuration description="Runs hello_world_test.">
  <option name="null-device" value="true" />
  <test class="com.android.tradefed.testtype.HostGTest" >
    <option name="module-name" value="hello_world_test" />
  </test>
</configuration>
```

The test configuration runs a gtest test (hello_world_test) on a host. The example test config can be auto-generated. Unless your test needs a special setup or cleanup, you can rely on auto test-config generation to create proper TF test configurations.

To configure a host-side gtest and enable auto test- config generation, set host_supported to true in Android.bp, as in hello_world_test.

For more information about writing a native test, see Adding a New Native Test Example.

JAR host tests
JAR (Java) host tests, such as JUnit, are tests that don't need to run on a device, and that provide code coverage of your Java project. Such tests can be configured to run on a test host by using the runner HostTest.

Sample TradeFed test configuration

```
<configuration description="Executes HelloWorldHostTest">
    <test class="com.android.tradefed.testtype.HostTest" >
        <option name="jar" value="HelloWorldHostTest.jar" />
    </test>
</configuration>
```

The test configuration runs a host-side JUnit test of HelloWorldHostTest. Note that the above test configuration can be auto-generated. Unless your test needs special setup or cleanup, rely on the auto test-config generation to create proper TradeFed test configuration.

For more details about how to write a JAR host test, refer to the JAR (Java) Host Tests page.

Isolated Java host tests
Deviceless Java tests can be run in an isolation environment with a slight performance cost. However, there are some major considerations to be made before choosing to use this environment.

This is the default runner used for Robolectric and JUnit unit tests
Tradefed supports only JUnit tests in the isolation environment.
Only statically linked dependencies are supported. No dependencies declared with lib are included on the classpath.
The isolation runner only puts the shim runner and your test jar on the classpath.
There is some amount of fixed overhead per test run executed with this runner.
Sample Tradefed test configuration (isolated)

```
<configuration description="Executes HelloWorldHostTest">
    <test class="com.android.tradefed.testtype.IsolatedHostTest" >
        <option name="jar" value="HelloWorldHostTest.jar" />
    </test>
</configuration>
```

Sample Soong configuration for autogeneration

Instead of manually creating the test config like above, Soong can autogenerate the config by using a declaration like this example.

```
java_test_host {
    name: "HelloWorldHostTest",

    test_options: {
        unit_test: true,
    },

    test_suites: ["general-tests"],

    srcs: ["test/**/*.java"],

    static_libs: [
        "junit",
    ],
}
```

Robolectric tests

Robolectric tests use the same runner as the isolated host tests, with a few special options.

The robolectric-resources option enables a few Robolectric-specific command line options to be passed into the subprocess as well as adds the tree build of android-all to the subprocess classpath. While the other two are best practices, this option is mandatory for running Robolectric tests with any success.

The java-folder option allows changing the Java runtime used by the subprocess. This is necessary due to Robolectric preferring particular Java versions that might not align with the host system's preferred JVM.

The exclude-paths option allows the subprocess runner to avoid loading particular modules at all, which is useful when a JAR comes with extraneous classes that could cause load errors. java. is a common exclusion, to avoid throwing SecurityException exceptions.

Sample Robolectric config

```
<configuration description="Executes a Sample Robolectric Test">
    <option name="java-folder" value="prebuilts/jdk/jdk9/linux-x86/" />
    <option name="exclude-paths" value="java" />
    <option name="use-robolectric-resources" value="true" />
    <test class="com.android.tradefed.testtype.IsolatedHostTest">
        <option name="jar" value="RobolectricExampleTest.jar" />
    </test>
</configuration>
```

Sample Soong configuration for Robolectric autogeneration

Instead of manually creating the test config like above, Soong can autogenerate the config by using a declaration like this example.

```
android_robolectric_test {
    name: "HelloWorldRoboTest",
    srcs: [
        "src/**/*.java",
    ],

    // Include the testing libraries
    static_libs: [
        "mockito-robolectric-prebuilt",
        "platform-test-annotations",
        "testng",
        "truth-prebuilt",
    ],

    instrumentation_for: "HelloWorldApp",
}
```

Python test

If the test logic is written in Python, use build type python_test_host to create a par file that can be run by TF PythonBinaryHostTest.

Sample TradeFed test configuration

```
<configuration description="Config to run atest unittests">
    <test class="com.android.tradefed.testtype.python.PythonBinaryHostTest" >
        <option name="par-file-name" value="atest_unittests" />
        <option name="test-timeout" value="2m" />
    </test>
</configuration>
```

Test suite setting

For the host-side test to be accessible by TF for a given build, set the test module **`test_suites`** setting to **`general-tests`**:

```
test_suites: ["general-tests"],
```

With this setting, the test is packaged to general-tests.zip on the test_suites target.

Here's a clear and concise breakdown of how to **write a host-side deviceless test** in **Trade Federation (Tradefed)** — including what it is, why you'd use it, and examples for each test type.

💡 What Is a Host-Side Deviceless Test?

A **host-side deviceless test** is a test that:

- Runs **entirely on the host (your PC or a test server)**.
- **Does not require** any Android device or emulator.
- Is useful for testing logic, libraries, scripts, or services **that don't depend on a physical device**.

✅ Why Use Host-Side Deviceless Tests?

Use them when you want to:

- Run **unit tests** on Java or native code.
- Test **business logic** or tools used in development pipelines.
- Validate host-side tooling (like parsers, log analysis tools, etc.).
- Get **fast test execution** (no device communication overhead).

🔪 Supported Types of Host-Side Deviceless Tests in Tradefed

1. Native (gtest) Host Tests

Example: Test a C++ library with gtest

- Runs natively on the host.
- Fast and no device needed.

📄 XML Config

```xml
xml

<configuration description="Runs hello_world_test.">
    <option name="null-device" value="true" />
    <test class="com.android.tradefed.testtype.HostGTest">
        <option name="module-name" value="hello_world_test" />
    </test>
</configuration>
```

🧱 Soong `Android.bp`

```
bp

cc_test_host {
    name: "hello_world_test",
    host_supported: true,
    srcs: ["hello_world_test.cpp"],
    test_suites: ["general-tests"],
}
```

---

2. Java (JAR) Host Tests

- For running plain **JUnit** tests.
- Use when testing Java logic that doesn't touch Android APIs.

📄 XML Config

```xml
<configuration description="Executes HelloWorldHostTest">
    <test class="com.android.tradefed.testtype.HostTest">
        <option name="jar" value="HelloWorldHostTest.jar" />
    </test>
</configuration>
```

🧱 Soong `Android.bp`

```bp
java_test_host {
    name: "HelloWorldHostTest",
    srcs: ["test/**/*.java"],
    static_libs: ["junit"],
    test_suites: ["general-tests"],
    test_options: {
        unit_test: true,
    },
}
```

---

3. Isolated Java Host Tests (JUnit/Robolectric)

- Test runs in an **isolated Java process**.
- No extra dependencies or classpath bloat.
- Preferred for **Robolectric** or **strict JUnit**.

📄 XML Config

```xml
<configuration description="Executes HelloWorldHostTest">
    <test class="com.android.tradefed.testtype.IsolatedHostTest">
        <option name="jar" value="HelloWorldHostTest.jar" />
    </test>
</configuration>
```

🧱 Soong `Android.bp`

```bp
java_test_host {
    name: "HelloWorldHostTest",
```

```
    srcs: ["test/**/*.java"],
    static_libs: ["junit"],
    test_suites: ["general-tests"],
}
```

4. Robolectric Tests

- Great for unit testing Android components without a real device.
- Runs in isolated host environment.

📄 XML Config

```xml
<configuration description="Executes Robolectric test">
    <option name="java-folder" value="prebuilts/jdk/jdk9/linux-x86/" />
    <option name="exclude-paths" value="java" />
    <option name="use-robolectric-resources" value="true" />
    <test class="com.android.tradefed.testtype.IsolatedHostTest">
        <option name="jar" value="RobolectricExampleTest.jar" />
    </test>
</configuration>
```

🧱 Soong `Android.bp`

```bp
android_robolectric_test {
    name: "HelloWorldRoboTest",
    srcs: ["src/**/*.java"],
    static_libs: [
        "mockito-robolectric-prebuilt",
        "platform-test-annotations",
        "testng",
        "truth-prebuilt",
    ],
    instrumentation_for: "HelloWorldApp",
}
```

5. Python Host Tests

- Use when your logic or scripts are written in Python.
- Runs a `.par` file packaged with Python and dependencies.

📄 XML Config

```xml
<configuration description="Config to run Python tests">
    <test class="com.android.tradefed.testtype.python.PythonBinaryHostTest">
        <option name="par-file-name" value="atest_unittests" />
        <option name="test-timeout" value="2m" />
```

```
    </test>
</configuration>
```

🧱 **Soong `Android.bp`**

```
bp

python_test_host {
    name: "atest_unittests",
    srcs: ["atest_unittests.py"],
    main: "atest_unittests.py",
    test_suites: ["general-tests"],
}
```

🧩 Important Settings

- Always add:

  ```
  bp

  test_suites: ["general-tests"]
  ```

  This makes your test discoverable by Tradefed and included in **`general-tests.zip`**.

✅ Summary Table

Test Type	Runner Class	Needs Device	Notes
Native (gtest)	**`HostGTest`**	❌	Fast, C++ gtests
Java JUnit (JAR)	**`HostTest`**	❌	Simple JUnit test
Isolated Java	**`IsolatedHostTest`**	❌	Preferred for Robolectric
Robolectric	**`IsolatedHostTest`** + options	❌	Android component testing
Python	**`PythonBinaryHostTest`**	❌	Python **`.par`** execution

Would you like a full **working sample** with source + XML config that you can plug into AOSP and run?

```
yes
```

Great! Let's now **simplify** everything about **host-driven tests in Trade Federation (Tradefed)**:

✅ What is a host-driven test?

A **host-driven test** is:

- A **Java test** that **runs on your computer** (host), **not on the device**.
- It **controls or interacts with a device** via ADB (e.g., installs APKs, reboots device, checks logs).
- You write it like a **JUnit4** test.

✅ Why not just use normal instrumentation tests?

Instrumentation tests **run *on* the Android device**, so:

- If you **reboot the device**, the test process dies.
- You **can't control other devices**, or do things **outside the app**.

That's where **host-driven tests help**:

- They **stay alive** even if the device reboots.
- Can **talk to multiple devices**.
- Can **set up and clean up** stuff before/after the test.

✅ Basic Structure:

Here's a minimal example:

```
java

@RunWith(DeviceJUnit4ClassRunner.class)
public class MyHostTest extends BaseHostJUnit4Test {

    @Before
    public void setUp() throws Exception {
        // Do setup before each test
    }

    @Test
    public void testDeviceConnection() throws Exception {
        Assert.assertNotNull(getDevice()); // Check device is connected
    }
}
```

✅ Why extend `BaseHostJUnit4Test`?

Because it gives you:

- **`getDevice()`** → access to the Android device.
- **`installPackage()`** → install APK.

- `runDeviceTests()` → run an instrumentation test from host.
- `uninstallPackage()` → remove app.
- `hasDeviceFeature()` → check device features.
- `getTestInformation()` → access device/build/test info.

✅ Extra Setup for Whole Class:

You can run some code **once before all tests**, using:

```java
@BeforeClassWithInfo
public static void beforeAllTests(TestInformation info) {
    // Use info.getDevice(), store test-wide values
}
```

✅ How to run it with Tradefed:

In the test configuration XML:

```xml
<test class="com.android.tradefed.testtype.HostTest">
    <option name="class" value="android.sample.cts.MyHostTest" />
</test>
```

✅ Types of actions you can test:

- Check whether an APK is installed.
- Install and uninstall an APK.
- Reboot the device and verify recovery.
- Interact with logcat or shell commands.
- Trigger app start and verify behavior.
- Run commands that modify system state.

Would you like a **full working example project** with this type of test and how to run it with Tradefed CLI?

Report metrics or data from a Tradefed test

bookmark_border
This page describes how to report metrics along with test results when writing a test in Tradefed.

The benefit of logging through Tradefed pipeline is to find your metrics along side your functional results. The logging of metrics can be done very naturally within tests, which makes it convenient for test writers to add more instrumentation.

DeviceTestCase - JUnit3 style
If your test extends DeviceTestCase in a JUnit3-style kind of test, you can call the method addTestMetric(String key, String value) from inside any test cases to report a metric. This can be called multiple times as long as the key is unique.

Example:

```
  public static class TestMetricTestCase extends DeviceTestCase {

    public void testPass() {
      addTestMetric("key1", "metric1");
    }

    public void testPass2() {
      addTestMetric("key2", "metric2");
    }
  }
```

If you want to log a file to be available in the result_reporters, you can call the method addTestLog(String dataName, LogDataType dataType, InputStreamSource dataStream) from inside any test cases to report a file to log.

Example:

```
  public static class TestLogTestCase extends DeviceTestCase {

    public void testPass() {
      try (InputStreamSource source = getDevice().getScreenshot()) {
        addTestLog("screenshot", LogDataType.PNG, source);
      }
    }
  }
```

TestCase - regular JUnit3 test
If you want to report metrics inside Tradefed from a regular JUnit3 TestCase class, it will need to

be converted to a MetricTestCase instead which is the exact same class with an extra method: addTestMetric(String key, String value)

DeviceJUnit4ClassRunner - JUnit4 style
If your JUnit4 style test is running with DeviceJUnit4ClassRunner, then you can also log metrics within a test case (inside @Test) to be reported by Tradefed. You will need to use TestMetrics rules to report your metrics.

Example:

```
@RunWith(DeviceJUnit4ClassRunner.class)
public static class Junit4TestClass {

    @Rule
    public TestMetrics metrics = new TestMetrics();

    @Test
    public void testPass5() {
        // test log through the rule.
        metrics.addTestMetric("key", "value");
    }

    @Test
    public void testPass6() {
        metrics.addTestMetric("key2", "value2");
    }
}
```

In order to report files, you will use the TestLogData rule to report it.

Example:

```
@RunWith(DeviceJUnit4ClassRunner.class)
public static class Junit4TestClass {

    @Rule
    public TestLogData logs = new TestLogData();

    @Test
    public void testPass5() {
        // test log through the rule.
        try (InputStreamSource source = getDevice().getScreenshot()) {
            logs.addTestLog("screenshot", LogDataType.PNG, source);
        }
    }
```

```
    }
```

IRemoteTest - pure Tradefed Test

If you are writing your own Tradefed Test class or runner you will implemement IRemoteTest and get a ITestInvocationListener through the run() method. This listener can be used to log metrics as follows:

```
    listener.testLog(String dataName, LogDataType type of data, InputStreamSource data);
```

Tradefed metrics collectors

Tradefed provides a dedicated metrics_collector object to collect metrics in parallel of the tests.

On the host side

BaseDeviceMetricCollector can be implemented to collect any metrics from the host-side and report them as part of the test invocation. A number of generic collectors are already available for different use cases, but we always welcome new contributions.

In order to specify the collector to be used in your Tradefed invocation, you simply need to add the object to your Tradefed XML configuration:

Example:

```
  <metrics_collector class="com.android.tradefed.device.metric.AtraceCollector">
     <option name="categories" value="freq"/>
  </metrics_collector>
```

Some currently existing collectors: * TemperatureCollector that collects the temperature periodically during the test run. * AtraceCollector that collects using 'atrace' for each test case.

On the device side

When running device-side tests (Instrumentations, UIAutomator tests, etc.), having a collector on the host-side collecting asynchronously might not be ideal. For example, a screenshot taken asynchronously will most likely miss the wanted screen and be useless.

In order to meet these use cases, a device-side version of our collectors exists and can be use in any 'AndroidJUnitRunner' instrumentation. BaseMetricListener can be implemented in order to automatically report metrics that are collected in a way fully compatible with the Tradefed reporting pipeline.

If you are using the 'AndroidJUnitTest' runner from Tradefed, you can simply specify the following command line option to have your collector running with your tests:

```
  --device-listeners android.device.collectors.ScreenshotListener
```

CAUTION: In order for the collector classes to be resolved at runtime, your instrumentation APK will most likely need to statically include them by adding to your makefile the following:

```
  LOCAL_STATIC_JAVA_LIBRARIES += collector-device-lib
```

Contributions to device-side collectors are also welcome.

Special consideration for suites

For suites like CTS that have a top-level configuration running some module configurations, there is no need to specify metrics_collector in each module configuration (AndroidTest.xml). It is actually forbidden.

To ensure the metric collection is applied equally to each module, only the top-level configuration (for example, cts.xml) can specify metrics_collector as explained above. These collectors will be applied and run against each module of the suite.

Collect device log files from a module

A setup is available in order for a device side test to notify that some files should be collected.

AndroidTest.xml can specify a collector that will look for file on the device and pull them.

```
  <metrics_collector class="com.android.tradefed.device.metric.FilePullerLogCollector">
     <!-- repeatable: Pattern of key of a FILE we listen on that should be pulled -->
     <option name = "pull-pattern-keys" value = "ScreenshotListener_.*" />

     <!-- repeatable: The key of the DIRECTORY to pull -->
     <option name = "directory-keys" value = "<example-key: /sdcard/atrace_logs>" />
  </metrics_collector>
```

By specifying these patterns and key, the collector if it sees the key will attempt to pull and log the associated file.

In order for these keys to be generated, a device-side test (instrumentation) should specify the file that should be logged. It is done in a similar manner as the host-side (described above).

Add the collector-device-lib to your test APK in the make files:

```
  LOCAL_STATIC_JAVA_LIBRARIES += collector-device-lib
```

Use the @rule we provide to log files:

```
   @RunWith(AndroidJUnit4.class)
   public static class Junit4TestClass {

      @Rule
      public TestLogData logs = new TestLogData();

      @Test
      public void testPass5() {
         // test log through the rule.
```

```
            File logFile = new File("whatever");
            logs.addTestLog("KEY", logFile);
        }
    }
```

The KEY name in the example above is the name under which the file will be reported. This is the name you should match in the FilePullerDeviceMetricCollector to get it automatically pulled. it should be a unique name.

NOTE: Once the file is pulled, FilePullerDeviceMetricCollector automatically cleans it from the device.

Where do I find the metrics?
It depends of the result_reporter specified in your XML configuration.

Here's a **simple explanation** of how to **report metrics or logs** from your Tradefed tests — whether you're testing on a **host** or a **device**.

🔧 What is "reporting metrics" in Tradefed?

When your test runs, it can output **extra data**, like:

- ✅ A status message (e.g., `"battery_level": "92%"`)
- 📊 A performance number (e.g., `"fps": "58.4"`)
- 🖼️ A screenshot or log file

These are called **metrics or logs**, and Tradefed lets you **attach them to the test result** so you can analyze them later.

✅ Types of Tradefed Tests That Can Report Metrics

1. JUnit3 Tests (DeviceTestCase)

Use this if your test extends `DeviceTestCase`.

```
java

public class MyTest extends DeviceTestCase {
    public void testSomething() {
        addTestMetric("boot_time", "3500ms");
    }
}
```

2. JUnit4 Tests (DeviceJUnit4ClassRunner)

Use **`@Rule`** with **`TestMetrics`** and **`TestLogData`**.

```
java

@RunWith(DeviceJUnit4ClassRunner.class)
public class MyJUnit4Test {

    @Rule public TestMetrics metrics = new TestMetrics();
    @Rule public TestLogData logs = new TestLogData();

    @Test
    public void testMetrics() {
        metrics.addTestMetric("frame_rate", "60fps");
    }

    @Test
    public void testScreenshot() throws Exception {
        try (InputStreamSource screenshot = getDevice().getScreenshot()) {
            logs.addTestLog("screenshot", LogDataType.PNG, screenshot);
        }
    }
}
```

---

3. Pure Tradefed Tests (IRemoteTest)

If you're building a custom test runner:

```
java

listener.testLog("logcat", LogDataType.TEXT, logStream);
listener.testMetrics("testName", "testClass", metricsMap);
```

🔪 Device-side Instrumentation Test + Metric Collection

If you're testing on the device and want to **automatically pull files or logs**:

1. Use **`@Rule`** with **`TestLogData`**
2. Name the file key clearly.

```
java

@Rule public TestLogData logs = new TestLogData();

@Test
public void testSaveFile() {
    File file = new File("/sdcard/mylog.txt");
    logs.addTestLog("MY_LOG_KEY", file);
}
```

Then in your **`AndroidTest.xml`**:

```xml
<metrics_collector class="com.android.tradefed.device.metric.FilePullerLogCollector">
    <option name="pull-pattern-keys" value="MY_LOG_KEY" />
</metrics_collector>
```

🗂️ Want to collect logs like logcat, atrace, or screenshots?

Use built-in collectors like:

Collector Name	What It Does
`AtraceCollector`	Collects system trace data during tests
`TemperatureCollector`	Monitors device temperature
`FilePullerLogCollector`	Pulls specific files from the device
`ScreenshotListener`	Takes screenshots during the test

📌 Where do metrics go?

Depends on how you run the tests:

- **Tradefed CLI** → outputs logs/metrics to the result directory.
- **Suite (e.g., CTS)** → stored in the module result folder.
- **Result Reporters** like **`XmlResultReporter`**, **`JSONReporter`**, or **`DashboardReporter`** decide where and how metrics are stored.

🚀 Summary

You want to...	Use this
Log custom data (e.g., "latency": "300ms")	**`addTestMetric()`** or **`TestMetrics`**
Save screenshots/log files	**`addTestLog()`** or **`TestLogData`**
Use built-in metrics	Add **`metrics_collector`** to XML
Pull logs/files from device	Use **`FilePullerLogCollector`** + **`TestLogData`**
Use metrics in JUnit4 tests	Use **`@Rule`** with **`TestMetrics`** and **`TestLogData`**

Want a working example project or XML config you can copy-paste?

confusing, explain with example

Sure! Let's go **step-by-step with a very simple working example** to show how to **report metrics and logs in Tradefed**, especially for **JUnit4 device tests**. ✅

🪛 Scenario:

You want to write a **JUnit4 test** that:

1. Logs a custom metric (`boot_time = 3500ms`)
2. Captures a **screenshot** from the device
3. Reports both back to Tradefed when the test runs

✅ Step 1: Write the JUnit4 Test Class

```
java

import com.android.tradefed.device.ITestDevice;
import com.android.tradefed.testtype.DeviceJUnit4ClassRunner;
import com.android.tradefed.device.DeviceTestCase;
import com.android.tradefed.result.InputStreamSource;
import com.android.tradefed.result.LogDataType;
import com.android.tradefed.testtype.junit4.DeviceTestRunConfiguration;
import com.android.tradefed.testtype.junit4.rules.TestLogData;
import com.android.tradefed.testtype.junit4.rules.TestMetrics;
import org.junit.Rule;
import org.junit.Test;
import org.junit.runner.RunWith;

import java.io.IOException;

@RunWith(DeviceJUnit4ClassRunner.class)
public class MyDeviceTest {

    @Rule public TestMetrics metrics = new TestMetrics();      // For reporting key-value
metrics
    @Rule public TestLogData logs = new TestLogData();         // For attaching
files/screenshots

    @Test
    public void testReportBootTimeAndScreenshot() throws Exception {
        // ✅ 1. Add a custom metric (string key-value)
        metrics.addTestMetric("boot_time", "3500ms");

        // ✅ 2. Capture screenshot and attach as log
        InputStreamSource screenshot = getDevice().getScreenshot();
        logs.addTestLog("boot_screen", LogDataType.PNG, screenshot);
    }

    private ITestDevice getDevice() throws Exception {
```

```
        return DeviceTestRunConfiguration.getDevice(); // Helper to get device in JUnit4 tests
    }
}
```

✅ Step 2: Run this test using Tradefed

```
sh

$ tradefed.sh
tf> run commandAndExit instrumentation com.android.test/.MyDeviceTest
```

This will:

- Run the test on the device
- Report **`boot_time = 3500ms`**
- Attach a screenshot under the name **`boot_screen`**

✅ Step 3: Output

After the test runs, you'll find:

- The **metric** (**`boot_time: 3500ms`**) printed in the console or saved in result files.
- The **screenshot** file (**`boot_screen.png`**) saved in the Tradefed result directory.

🔄 Optional: Use Metric Collector to auto-pull files

If your test writes a log file to **`/sdcard`**, you can tell Tradefed to automatically **collect that file** using XML config:

1. In test code:

```
java

@Rule public TestLogData logs = new TestLogData();

@Test
public void testSaveFile() {
    File myLog = new File("/sdcard/log.txt");
    logs.addTestLog("MY_LOG_KEY", myLog);
}
```

2. In XML config:

```xml
<metrics_collector class="com.android.tradefed.device.metric.FilePullerLogCollector">
    <option name="pull-pattern-keys" value="MY_LOG_KEY" />
</metrics_collector>
```

Tradefed will automatically pull **`/sdcard/log.txt`** and include it in results.

🧠 In Summary

Feature	How to Do It
Add custom metric	**`metrics.addTestMetric("key", "value")`**
Add screenshot/log	**`logs.addTestLog("name", LogDataType, ...)`**
Collect device files	Use **`FilePullerLogCollector`** in XML

Would you like me to give you a **full Android test project structure** with this code so you can run it directly?

details.

Disable automatic retry
Use the following option:

--retry-strategy NO_RETRY
Retry failures
To retry test failures, use the following options:

--retry-strategy RETRY_ANY_FAILURE --max-testcase-run-count X
This will retry the failure until it passes or until the max number of retries is reached, whichever comes first.

Note: max-testcase-run-count counts the number of run attempts, meaning in case of a crash, it is possible for a test case to not have been executed max-testcase-run-count times.
Iterations
To re-run tests for a number of time, the following options can be used:

--retry-strategy ITERATIONS --max-testcase-run-count X
What do the results look like?

> Result reporters by default will receive aggregated results of all attempts.
>
> For example: a Fail and a Pass for RETRY_ANY_FAILURE will result in an aggregated Pass since the retry managed to clear the failure.
>
> It is possible for reporters to receive the non-aggregated results. To do so, they need to extend the ISupportGranularResults interface that declares support for the granular (non-aggregated) results.
>
> Implementation details
> To enable auto-retry to rerun failures at the test case level, implement ITestFilterReceiver.
>
> If you can't implement ITestFilterReceiver, you can implement IAutoRetriableTest to manually control how rerun operates. InstalledInstrumentationsTest is an example implementation of that interface.

Let's break this down **simply and clearly** so you understand each part of **Tradefed retry logic** and **how to use it in real test runs**.

🔁 Why Retry?

Tradefed allows you to **automatically retry test cases** that fail due to:

- **Flaky behavior**
- **Device crashes**
- **Intermittent failures**

🔧 How to Control Retry

✅ 1. Disable Retry

If you don't want any automatic retry:

```
sh

--retry-strategy NO_RETRY
```

> 📌 This means even if a test fails or crashes, **Tradefed won't try again**.

✅ 2. Retry Failures Only

```
sh

--retry-strategy RETRY_ANY_FAILURE --max-testcase-run-count 3
```

This means:

- Retry a failed test **up to 3 times**
- If it passes during any retry, it's considered a **Pass** overall

  Example:

  - First run: testX → ❌ Fail
  - Retry 1: testX → ✅ Pass
  - Aggregated result = ✅ Pass

✅ 3. Repeat Tests Fixed Number of Times

```
sh

--retry-strategy ITERATIONS --max-testcase-run-count 3
```

This means:

- Run every test **exactly 3 times**, even if they pass
- This is useful for detecting flaky or timing-sensitive behavior

  Example:

  - testX → ✅
  - testX → ✅
  - testX → ✅
    Result: All 3 results recorded.

📋 What Do the Results Look Like?

- By default, **only aggregated results** are shown
- So if a test failed 2 times but passed on the 3rd:
  - Final result: ✅ PASS
  - Failures are hidden unless logging is customized

🧩 Advanced: Granular Results

- If you're building a **custom reporter** (e.g., XML, dashboard), you can get all retry attempts by implementing:

```
java

ISupportGranularResults
```

This interface allows your reporter to receive **all individual results**, not just the final pass/fail.

🔌 Integration: How to Support Retry in Your Test Class

1. Automatically retry test cases?

Implement this interface:

```
java

ITestFilterReceiver
```

- Tradefed will automatically filter and rerun only the failed tests.

2. Manually control retries?

Implement:

```
java

IAutoRetriableTest
```

- You control what happens during retries.
- Useful when you're writing custom logic for test flows.

  ✅ Example: **`InstalledInstrumentationsTest`** implements this to manage retries intelligently.

💡 Summary Table

Use Case	Command
Don't retry anything	**`--retry-strategy NO_RETRY`**
Retry only failures	**`--retry-strategy RETRY_ANY_FAILURE --max-testcase-run-count 3`**
Run all tests 3 times	**`--retry-strategy ITERATIONS --max-testcase-run-count 3`**

Use Case	Command
Want detailed logs of retries	Implement **`ISupportGranularResults`** in reporter
Want to retry only specific tests	Implement **`ITestFilterReceiver`**
Want to manually control retry	Implement **`IAutoRetriableTest`**

Would you like a **real command line example** with a test APK to see how retries work on an actual device?

yes

Great! Here's a **real working example** of how to run an instrumentation test using **Tradefed** with **retry logic** on a connected Android device.

✅ Step-by-step Example: Retry Instrumentation Test

🔪 Test Setup

Assume you have an instrumentation APK installed:

- **Package**: `com.example.myapp.test`
- **Runner**: `androidx.test.runner.AndroidJUnitRunner`

You want to run tests from this APK using **Tradefed**, and retry if a test fails.

🗂️ Command Line: Retry Failed Tests (Max 3 Attempts)

```
bash

tradefed.sh run instrumentation \
  --test-package com.example.myapp.test \
  --class com.example.myapp.ExampleInstrumentedTest \
  --runner androidx.test.runner.AndroidJUnitRunner \
  --retry-strategy RETRY_ANY_FAILURE \
  --max-testcase-run-count 3
```

🔍 What This Does

Option	Meaning
`run instrumentation`	Run an instrumentation test

Option	Meaning
`--test-package`	Specifies test APK package
`--class`	Specific test class to run
`--runner`	The instrumentation runner class
`--retry-strategy RETRY_ANY_FAILURE`	Retry if the test fails
`--max-testcase-run-count 3`	Try a test max 3 times

💥 What Happens on Failure?

If your test (e.g., `ExampleInstrumentedTest.testLogin()`) fails:

- Tradefed will rerun it
- If it passes on retry, final result = ✅ PASS
- If it fails 3 times, final result = ❌ FAIL

📦 Optional: Run All Tests 5 Times to Check Flakiness

```
bash

tradefed.sh run instrumentation \
  --test-package com.example.myapp.test \
  --runner androidx.test.runner.AndroidJUnitRunner \
  --retry-strategy ITERATIONS \
  --max-testcase-run-count 5
```

This will run **every test 5 times** no matter what.

📤 Where Are Results?

By default, test results are stored in:

```
plaintext

/tradefed_logs/
```

or shown in terminal as:

```
plaintext

[TestResultReporter] Result: PASS
```

To export to XML or other formats, you can enable result reporters:

```
bash

--template:map test-config=com/example/myapp/test.xml
```

Would you like me to show how to write a **`config.xml`** to automate this, or how to write your own custom test retry logic in Java with **`ITestFilterReceiver`**?

Retry isolation

bookmark_border
See the Automatic Test Retry documentation to understand how the retry feature works.

When executing retries, if the device has entered into a bad state that does not allow tests to succeed anymore. Use the isolation feature restore the device into a fresh initial state and allow tests to run and succeed.

Enable the feature
The retry isolation can be enabled with the follow options in addition to the retry options:

--retry-isolation-grade <FULLY_ISOLATED|REBOOT_ISOLATED>
or disabled with:

--retry-isolation-grade NOT_ISOLATED
The feature is disabled by default.

Different grade of isolation
The isolation grade represents the extent of the isolation we want to configure between retries:

FULLY_ISOLATED will completely wipe & resetup the device by rerunning the configured target_preparers of the configuration
REBOOT_ISOLATED will reboot the device
Note on Compatibility Test Suite (CTS)
Android partner runs of CTS currently do not support FULLY_ISOLATED

What do the results look like?
Result reporters receive a marker that a given module or run was running in isolation and can elect to display the marker in their report.

At the module level the module-isolated property will be set to its isolation grade.
At the run level the run-isolated property will be set to its isolation grade.

> For example: the first module running after being isolated will be reported as module-isolated:FULLY_ISOLATED.
>
> A test run in isolation provides a strong signal. Be it pass or fail, the test owner should have high confidence that no bad state from a previous test was left on the device.

Great question! Let's break down **Retry Isolation** in **Tradefed (TF)** step by step, with **clear examples** so it's not confusing.

✅ What is Retry Isolation in Tradefed?

When tests **fail**, Tradefed can **automatically retry** them.

But... what if the **device is in a broken state** after the first failure (e.g., app crash, ANR, or leftover files)?

> 🗝️ **Retry Isolation** ensures that the **retries happen in a clean, controlled environment**, so they aren't influenced by side effects from the previous attempt.

🧱 Isolation Levels

There are **three levels** you can choose using:

```
bash

--retry-isolation-grade <GRADE>
```

Grade	Description
`NOT_ISOLATED`	Default. **No cleanup** between retries. Retry uses the device **as-is**.
`REBOOT_ISOLATED`	Reboot the device before retrying. Medium cleanup.
`FULLY_ISOLATED`	Reboot + **re-run target preparers** to set up the device from scratch. 🔁

> 📌 `FULLY_ISOLATED` is like a fresh install + test rerun (like starting over).

📌 When is it Useful?

- Device enters a **bad state**: system crash, unresponsive, data left behind
- Test passes **only after clean reboot**
- Flaky test that needs a **consistent state**

🗂️ Real Example

🎯 Goal: Retry a failed test, but reboot the device first

```
bash

tradefed.sh run instrumentation \
  --test-package com.example.myapp.test \
  --class com.example.myapp.MyTestClass \
  --runner androidx.test.runner.AndroidJUnitRunner \
  --retry-strategy RETRY_ANY_FAILURE \
  --max-testcase-run-count 2 \
  --retry-isolation-grade REBOOT_ISOLATED
```

💥 What this does:

- If **`MyTestClass`** fails
- Tradefed reboots the device
- Then **retries** the test on a clean boot

🧪 FULLY_ISOLATED Example (For in-depth cleanup)

```
bash

tradefed.sh run instrumentation \
  --test-package com.example.myapp.test \
  --runner androidx.test.runner.AndroidJUnitRunner \
  --retry-strategy RETRY_ANY_FAILURE \
  --max-testcase-run-count 3 \
  --retry-isolation-grade FULLY_ISOLATED
```

This will:

- Retry failed test
- **Wipe device setup**
- Re-run **target_preparers** (e.g., installs, environment setup)
- Run the test again

⚠️ **Note**: CTS **does not support `FULLY_ISOLATED`**.

📊 What Do Reports Show?

If isolation was used, you'll see:

- **`module-isolated: FULLY_ISOLATED`**
- **`run-isolated: REBOOT_ISOLATED`**

These markers help test reviewers know that:

> "This test was retried in a clean state."

🧠 Summary

Option	Use When...
`NOT_ISOLATED` (default)	You trust the device is okay between retries
`REBOOT_ISOLATED`	You want a fresh OS state without full reinstall
`FULLY_ISOLATED`	You want a **factory-reset-style** retry environment

Would you like a real XML config that uses **`FULLY_ISOLATED`** or an example showing how to create custom **`target_preparers`** for isolation?

Automated log on failure collection

bookmark_border
When debugging tests, a set of logs is always needed to get a basic picture of the failure and the device under test. Sources include: Logcat, Tradefed host log, screenshot, etc.

In order to make it generic and painless for any test writer to get those logs, Tradefed has a built-in mechanism to help collecting them.

Configuration
To automatically collect some logs on failure, you can add the following option to your Tradefed command line:

--auto-collect LOGCAT_ON_FAILURE
or
--auto-collect SCREENSHOT_ON_FAILURE
To see the full list of possible values, checkout AutoLogCollector

For convenience, logcat and screenshot each have a direct flag:

--logcat-on-failure
and
--screenshot-on-failure
Note on suite modules (AndroidTest.xml)

> Modules cannot direcly specify this option in the AndroidTest.xml, but they can use a module controller instead.

In **simple terms**, here's what **"Automated log on failure collection"** in **Tradefed** means:

🔍 What is it?

When a test fails, you usually want to **know why**. You might need:

- **logcat** output (system logs from the Android device)
- A **screenshot** of the screen at the time of failure
- Tradefed's **host log**
- Other debugging info

To **make your life easier**, Tradefed can **automatically collect these logs** when a test fails. No need to manually capture them every time.

✅ How to enable it

You add special options to your Tradefed command line when running tests:

🔸 **Collect logcat:**

```
csharp

--logcat-on-failure
```

or

```
cpp

--auto-collect LOGCAT_ON_FAILURE
```

🔸 **Take screenshot:**

```
csharp

--screenshot-on-failure
```

or

```
cpp

--auto-collect SCREENSHOT_ON_FAILURE
```

These options tell Tradefed:

> "Whenever a test fails, collect this info for me automatically."

📌 Why is this useful?

Because when a test fails:

- You instantly get logs and a screenshot
- You don't need to rerun just to figure out what went wrong
- Makes **debugging faster and easier**

🧩 Extra Note (for advanced users)

If you're writing **suite-based tests** using `AndroidTest.xml`, you **can't directly** add these options in the XML. Instead:

- You use a **ModuleController** to set them up programmatically.

🧪 Example

Imagine you're running instrumentation tests like this:

```
bash

run commandAndExit adb-shell am instrument ...
```

With failure logging enabled:

```
bash

run commandAndExit ... --logcat-on-failure --screenshot-on-failure
```

If the test crashes or fails:
✅ Logcat and screenshot are saved automatically
🗂️ You'll find them in the results directory

Let me know if you'd like a **real example** or how to **view the collected logs**!

Dry run a configuration

bookmark_border
It can be convenient to dry-run a Trade Federation (TF) configuration to ensure that TF is properly configured.

There are two different mechanisms in TF for this.

General configuration
The dry-run command checks that a configuration is parsed correctly and is runnable. The command does not run TF so this validation is limited to the TF configuration syntax.

Configuration for test suites
For the test suite configurations, such as CTS, the --collect-tests-only command is available. It runs everything except the test cases, which are enumerated. However, it does run all the other setup steps, like installing APKs.

This is a convenient option to ensure that a configuration runs as expected without actually running the tests.

Support for collect-tests-only
The support for the collect-tests-only is exposed via ITestCollector.

A test runner that implements this interface enumerates its test cases instead of running them.

InstrumentationTest is an example implementation of that interface.

Let's break down **"Dry run a configuration"** in Tradefed (TF) into **simple and clear** parts:

🤔 What is a "dry run"?

A **dry run** means:

> "Run the configuration/setup, but **don't actually run the tests**."

This is helpful when you want to:

- Make sure your config is valid ✅
- Confirm things like APKs are installing, devices are detected, etc. ✅
- But **skip the actual test execution** ❌

✅ Types of Dry Runs in Tradefed

1. General Dry Run

- Use this to **check if a configuration is valid** (e.g. correct syntax, correct XML structure).
- Command:

```
bash

run dry-run <configuration>
```

- ✅ Checks syntax
- ❌ Doesn't do anything on the device
- ❌ Doesn't install APKs
- ❌ Doesn't run tests

---

2. Suite Dry Run (Test Collector Mode)

- For **CTS, GTS, VTS**, etc., use:

```
bash

run cts --collect-tests-only
```

- ✅ Runs setup: device detection, target preparers, APK installation
- ✅ Enumerates test cases (lists them)
- ❌ **Does not run** any test cases

This is **very useful before large test runs**, to ensure:

- Your suite is working
- All setup steps complete
- You'll know **what tests will be run**

🧠 How is it implemented?

Test modules support this via:

```
java

ITestCollector
```

This interface tells Tradefed:

> "If collect-tests-only is enabled, just list test cases—don't execute them."

🔸 **Example:**

`InstrumentationTest` (used for app-based tests) implements `ITestCollector`.

🪛 Example Use Case

Let's say you're about to run CTS:

```
bash

run cts --module Camera
```

Instead, first check:

```
bash

run cts --module Camera --collect-tests-only
```

What happens?

- Your device will be detected
- Camera APKs will be installed
- Test cases will be **listed but not executed**
- If anything fails (e.g. install fails), you'll catch it **before wasting time** running the full test

---

✅ Summary

Mode	What it checks	Runs setup	Runs tests
`dry-run`	Validates config syntax	❌	❌
`--collect-tests-only`	Checks full setup, lists tests	✅	❌

Let me know if you want to try a specific config or command.

> Run instrumentation tests from existing APKs
>
> bookmark_border

These instructions assume you have the Trade Federation package available locally; if not, follow the download instructions to obtain it.

Then use the following command to install the APK of instrumentation tests, execute the tests, and display the tests that are running:

./tradefed.sh run instrumentations --apk-path <path of your apk>
With output resembling:

07-17 10:55:32 D/InvocationToJUnitResultForwarder: Starting test:
android.animation.cts.ValueAnimatorTest#testOfArgb
07-17 10:55:33 D/InvocationToJUnitResultForwarder: Starting test:
android.animation.cts.ValueAnimatorTest#testIsRunning
07-17 10:55:34 D/InvocationToJUnitResultForwarder: Starting test:
android.animation.cts.ValueAnimatorTest#testGetCurrentPlayTime
07-17 10:55:35 D/InvocationToJUnitResultForwarder: Starting test:
android.animation.cts.ValueAnimatorTest#testStartDelay
07-17 10:55:35 I/InvocationToJUnitResultForwarder: Run ended in 2m 20s
You can optionally specify --serial <device serial number> to run against a given device. The serial number of your device can be obtained using adb devices.

See the Testing through Tradefed section for more details about Tradefed executions.

Here's a **simple guide** to help you **run instrumentation tests from existing APKs using Tradefed**:

✅ What you need

- Trade Federation (Tradefed) downloaded and built (or use from AOSP source tree).
- An **APK file that contains instrumentation tests**, i.e., built with an `androidTest` target.
- An **Android device** connected via ADB.

🔧 Basic Command

```
bash

./tradefed.sh run instrumentations --apk-path <path-to-your-instrumentation-apk>
```

This will:

- Install the APK on the connected device
- Find and execute all `@Test` methods in the instrumentation test APK

- Print test results to the console

🎯 Optional: Specify Device

If multiple devices are connected, use:

```
bash

--serial <device-serial>
```

You can get your device's serial number by running:

```
bash

adb devices
```

Example:

```
bash

./tradefed.sh run instrumentations --apk-path
out/target/product/vsoc_x86/data/app/MyAppTests.apk --serial emulator-5554
```

🖥️ Sample Output

Tradefed will show output like:

```
bash

Starting test: com.example.app.ExampleInstrumentedTest#useAppContext
Starting test: com.example.app.ExampleInstrumentedTest#checkButtonClick
Run ended in 1m 10s
```

🔍 What counts as an Instrumentation APK?

- It's an APK generated by building **`androidTest`**, e.g. from:

  ```
  bash

  ./gradlew assembleDebugAndroidTest
  ```

- It has an **instrumentation test runner** declared in **`AndroidManifest.xml`**, like:

```xml
<instrumentation
    android:name="androidx.test.runner.AndroidJUnitRunner"
    android:targetPackage="com.example.app"
    android:label="Test runner" />
```

💡 Other Useful Flags

Option	Description
`--test-filter`	Run only specific test classes or methods
`--install-clean`	Uninstalls existing APKs before running
`--package`	Specify test package name if not auto-detecting
`--runner`	Use a specific test runner class

📦 Example: Full command

```bash
./tradefed.sh run instrumentations \
  --apk-path /tmp/mytests.apk \
  --serial emulator-5554 \
  --test-filter com.example.MyTestClass#testSomething \
  --runner androidx.test.runner.AndroidJUnitRunner
```

Let me know if you'd like help building the test APK, filtering specific tests, or integrating this into CTS or VTS modules.

Write a shell test in Trade Federation

bookmark_border
This page describes how to write a shell test. If a test can be done with a shell script, you can run it with ExecutableHostTest. ExecutableHostTest supports test filtering and test sharding.

Write a shell test
Your main shell script runs on the host. Other scripts or binaries to run on the device and the data files required must be pushed to the device first. This can be done as part of the main script or using PushFilePreparer.

If the test has device binary modules that should be installed alongside the test, specify them with data_device_bins property.

```
sh_test {
    name: "module-name",
    ...
    data_device_bins: ["target-name"],
}
```

Shell test ziptool-tests is such an example.

The test result for each test is based on the exit status of your main script. A test can timeout if per-binary-timeout has been specified.

Configure a shell test
In Tradefed XML configuration, shell tests are run through the ExecutableHostTest runner.

```
<test class="com.android.tradefed.testtype.binary.ExecutableHostTest" >
    <option name="binary" value="your-test-script.sh" />
    <option name="relative-path-execution" value="true" />
    <option name="per-binary-timeout" value="15m" />
</test>
```

Set the main script for the test as the value of the binary option. Check out other ExecutableHostTest options.

If the script requires root privilege, add RootTargetPreparer to the configuration.

Here's a **simple, complete guide to writing and configuring a shell test in Trade Federation (Tradefed)** using **`ExecutableHostTest`**.

✅ Why Write a Shell Test in Tradefed?

Shell tests are useful when:

- You want to **quickly validate device behavior** using **`adb shell`** commands.
- You need **custom automation logic** that doesn't need Java instrumentation.
- You want to test a **device binary or script** installed on the device.

🛠️ 1. Create the Shell Script

This is your **host-side script** that will:

- Optionally push files to the device.
- Run device-side binaries via **`adb shell`**.

- Return **exit code 0 = pass**, anything else = fail.

📄 `your-test-script.sh`

```
bash


#!/bin/bash

# Optional: push a test binary to device
adb push mytest.sh /data/local/tmp/
adb shell chmod +x /data/local/tmp/mytest.sh

# Run the test
adb shell /data/local/tmp/mytest.sh

# Get exit code from device test script
RESULT=$?
if [ $RESULT -eq 0 ]; then
  echo "Test passed"
else
  echo "Test failed with code $RESULT"
fi

exit $RESULT
```

Make it executable:

```
bash


chmod +x your-test-script.sh
```

📦 2. Put Script in a Shell Module (Android.bp)

```
python


sh_test {
    name: "my-shell-test",
    src: ["your-test-script.sh"],
    test_suites: ["general-tests"],
}
```

Optional:
If your test depends on a device binary, declare it:

```
python


sh_test {
    name: "my-shell-test",
    src: ["your-test-script.sh"],
    data_device_bins: ["my-device-binary"],
    test_suites: ["general-tests"],
}
```

⚙️ 3. Tradefed XML Config

Create a test config XML under **`test/configs/my_shell_test.xml`**:

```
xml

<configuration description="Runs my shell test script">
    <test class="com.android.tradefed.testtype.binary.ExecutableHostTest">
        <option name="binary" value="your-test-script.sh" />
        <option name="relative-path-execution" value="true" />
        <option name="per-binary-timeout" value="5m" />
    </test>
</configuration>
```

If the test requires **`adb root`**, add:

```
xml

<target_preparer class="com.android.tradefed.targetprep.RootTargetPreparer"/>
```

🔪 4. Run the Test

Once your build has included the test (appears in **`out/host/linux-x86/bin/`**):

```
bash

./tradefed.sh run config my_shell_test.xml
```

Or filter and shard:

```
bash

./tradefed.sh run commandAndExit \
    --template:map test=your-test-script.sh \
    --template:map shard-count=2
```

✅ Summary

Component	Purpose
`sh_test` module	Declares and builds the test
`ExecutableHostTest`	Tradefed runner that executes shell scripts on the host
`adb shell` inside script	Executes commands on the Android device
XML config	Describes how to run the shell test
Exit code of script	Determines pass/fail status

Let me know if you want a working example with a device-side script that checks file permissions, CPU info, or something else.

> Global filters in Trade Federation
>
> bookmark_border
> Global filters is a way to provide test filters to Tradefed that will apply to the invocation automatically without having to worry about whether a test supports it or not.
>
> It will be inop if the configuration doesn't support filtering, but provide the benefit of not throwing a configuration exception like it's usually the case in Tradefed. This allows a service with minimal knowledge about Tradefed to easily generate and provide filter flags generically.
>
> Syntax
>
> --global-filters:include-filter [abi] module-name [test-class][#method-name]
>
> --global-filters:exclude-filter [abi] module-name [test-class][#method-name]
> Where module-name is the only mandatory part to be provided.
>
> For example:
>
> --global-filters:include-filter moduleA : will only run the module
> --global-filters:include-filter moduleA class#method : will only run the matching method
> The option can be repeated if more filters are needed. Exclusion has priority over inclusion.
>
> Implementation
> Source code

In **Trade Federation (Tradefed)**, *Global Filters* provide a way to apply test filtering universally — across all modules — without requiring individual modules to explicitly support them.

🔧 Why use Global Filters?

Normally, filtering (like `include-filter` or `exclude-filter`) only works if the test runner implements filtering logic (like `ITestFilterReceiver`). If the test doesn't support filtering, it may throw a configuration error.

Global Filters solve this by:

- Not throwing exceptions if the test doesn't support filtering.
- Applying filters silently to applicable tests.

✅ Syntax

```
python

--global-filters:include-filter [abi] module-name [test-class][#method-name]
--global-filters:exclude-filter [abi] module-name [test-class][#method-name]
```

- **`abi`** is optional (like **`arm64-v8a`**)
- **`module-name`** is **mandatory**
- **`test-class`** and **`#method-name`** are optional refinements

💡 Examples

1. Run only a specific module:

```
bash

--global-filters:include-filter CtsGraphicsTestCases
```

2. Run a single test method:

```
bash

--global-filters:include-filter CtsGraphicsTestCases android.graphics.cts.BitmapTest#testCreate
```

3. Exclude a method:

```
bash

--global-filters:exclude-filter CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testBadInput
```

4. Include multiple modules/tests:

```
bash

--global-filters:include-filter CtsGraphicsTestCases
--global-filters:include-filter CtsAppTestCases android.app.cts.ActivityTest
```

5. Specify ABI (optional):

```
bash

--global-filters:include-filter arm64-v8a CtsGraphicsTestCases
```

🚫 Exclusion vs Inclusion

- Exclude filters **take priority** over include filters.
- If a method is both included and excluded, **it will not run**.

🧠 Internally...

When you use `--global-filters`, Tradefed checks if the test runner supports `ITestFilterReceiver`. If it does, filters are applied. If not, they are ignored **gracefully** (no crash).

📌 Summary

Option	Description
`--global-filters:include-filter`	Selects tests/modules to run
`--global-filters:exclude-filter`	Omits specific tests/modules
Applies silently	No crash if test doesn't support filtering
Exclude > Include	Exclusions take precedence over inclusions

Let me know if you want a practical example using `run cts` or `run commandAndExit`.

> yes

Here's a **practical example** of using **global filters** with a CTS (Compatibility Test Suite) run in Trade Federation:

🔪 Scenario:

You only want to run a **single test method** from a specific CTS module (e.g., `CtsGraphicsTestCases`) — say, `android.graphics.cts.BitmapTest#testCreate`.

✅ Command using Global Filter:

```
bash


run cts \
  --module CtsGraphicsTestCases \
  --global-filters:include-filter CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testCreate
```

This tells Tradefed:

- Load the **`CtsGraphicsTestCases`** module
- Filter down to only one specific test method in that module
- Even if the test doesn't explicitly implement **`ITestFilterReceiver`**, it won't crash — it will just skip if not applicable

---

⛔ Excluding a known bad test (e.g., flaky one):

```
bash


run cts \
  --module CtsGraphicsTestCases \
  --global-filters:exclude-filter CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testBadInput
```

This will run all tests in **`CtsGraphicsTestCases`** **except** the excluded one.

📎 Multiple Filters Together:

```
bash


run cts \
  --module CtsGraphicsTestCases \
  --global-filters:include-filter CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testCreate \
  --global-filters:exclude-filter CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testBadInput
```

Even though you're including **`testCreate`**, if you also excluded it accidentally, the **exclusion will win** and it won't run.

🧠 Optional ABI Filtering:

```
bash


run cts \
  --global-filters:include-filter arm64-v8a CtsGraphicsTestCases
android.graphics.cts.BitmapTest#testCreate
```

Runs the test only for devices with the **`arm64-v8a`** ABI.

Would you like to try this with **`run commandAndExit`** or in an actual **`.xml`** config file too?

Test through a suite

bookmark_border
1. Set up suite
Tests running through a suite differs from a standard test running directly through Tradefed test runner.

See Set up Suite for definitions and instructions.

2. Pass options and filters to suite and modules
Various command lines and arguments can be used to pass options and filters to the suite-level or the module-levels.

See Pass options and filters to the suite and the modules for instructions.

3. System Status Checker
System Status Checker is a component running between each module that can check and clean up some states that the module might have forgotten to clean, for example a module leaving a keyguard on.

See System status checker for implementation and use.

4. AndroidTest.xml structure
Each AndroidTest.xml file represents one test module. This section describes the structure of an AndroidTest.xml configuration file, how they differ from regular Tradefed configuration, and what is allowed and not allowed.

See AndroidTest.xml structure for allowed tags and an example configuration.

5. Tune sharding configuration
This section explains how to tune sharding aspects of your module configuration and describes the benefits associated with each of them:

Declare your module shardable or not
Tune the possible number of shards for your module
See Tune AndroidTest.xml modules for details.

6. Controller
Module controllers are special objects that can be specified in AndroidTest.xml to alter some

aspect of the module's behavior. For example, you may completely skip the module if some conditions are not met.

See Module controllers for implementation and logging.

7. Retry
Suite Retry allows you to re-run a previous invocation of failed tests in order to rule out flakiness or poor isolation that may have caused the first failure.

See Suite Retry for examples and use.

8. Multi devices suites
Newer releases now support multi-devices suites and modules which allow for a test to coordinate between multiple devices in its test flow.

See Multi-devices suites for more details.

Here's a **simple explanation** of how **test execution through a suite** (like CTS, VTS, etc.) works in **Trade Federation (Tradefed)**, breaking down each of the 8 components from your summary:

✅ 1. Set up suite

- **Suites** (e.g., CTS, VTS) are pre-defined *collections of modules*.
- Each module has its own config (`AndroidTest.xml`).
- You run a suite using `run cts` or `run vts`, not `run commandAndExit`.

Example:

```
bash

run cts --module CtsAppTestCases
```

✅ 2. Pass options and filters

- You can apply options globally (for the entire suite) or specifically (for one module or test).
- Use flags like:
  - `--module`, `--test`, `--include-filter`, `--exclude-filter`
  - Or more generally: `--global-filters:include-filter`

Example:

```
bash

run cts --module CtsAppTestCases --include-filter android.app.cts.ActivityTest#testLaunch
```

✅ 3. System Status Checker

- Checks for bad system state *between modules*.
- Example: if a test left the device in a locked state or airplane mode, status checkers will reset it.
- Default ones: **`KeyguardStatusChecker`**, **`BatteryStatusChecker`**, etc.
- Helps keep test runs isolated and consistent.

✅ 4. AndroidTest.xml

- Each module in a suite has its own **`AndroidTest.xml`**.
- It defines:
  - Test class (e.g., **`InstrumentationTest`**)
  - APKs to install
  - Option parameters
- It's more restricted than a regular Tradefed XML.
  - Example snippet:

```
xml

<configuration description="CtsAppTestCases">
  <option name="test-suite-tag" value="cts"/>
  <test class="com.android.tradefed.testtype.InstrumentationTest">
    <option name="package" value="android.app.cts"/>
  </test>
</configuration>
```

✅ 5. Tune sharding configuration

- Suites support **sharding** (splitting work across devices or threads).
- Can define:
  - Whether a module supports sharding
  - Max number of shards per module

Benefits:

- Faster test execution across many devices or processes.

✅ 6. Controller

- **Module Controllers** decide whether a module should run.
- For example, a Bluetooth test could be skipped if the device lacks Bluetooth.

In XML:

```
xml

<module_controller class="com.android.tradefed.testtype.suite.module.Sdk31ModuleController"/>
```

✅ 7. Retry

- **Suite Retry** lets you re-run *only failed tests* from a previous session.

Example:

```
bash

run cts --retry 1234567890 --retry-type FAILED
```

Where **`1234567890`** is the previous invocation ID.

✅ 8. Multi-device suites

- Newer frameworks allow one test module to use **multiple devices** at the same time.
- Useful for:
  - Bluetooth pairing
  - Wi-Fi Direct
  - Casting tests
- Devices are selected using **`--device`** flags and handled via test code.

Would you like a **visual diagram** or **real XML + test file example** for one of these steps?

> Set up suites
>
> bookmark_border
> A suite in Tradefed refers to a setup where several tests are running under a common test

runner that drives the overall execution.

In Tradefed, suites are driven through the ITestSuite class, which lets tests be added and removed independently of how they are run.

Definitions
Suite: Set of test modules configured to run under a similar top-level setup to report their results under a single invocation.
Top-level setup: Setup applied to the devices before running any of the test modules.
Main configuration: The suite-level Tradefed XML configuration that describes which modules should run and which top-level setup should be used.
Module-level setup: Setup applied to the devices right before running the module. These are also known as module-specific setups.
Module configuration: Refers to the AndroidTest.xml Tradefed XML configuration that describes the modules and which module-level setup should be done.
Module: Test unit composed of a setup step (module-level setup), a test execution step and a tear down step.
Intra-module retry: Automatic retry done by the harness inside the module.
Suite retry: Full rerun of the suite's previously failed tests.
ITestSuite structure
ITestSuite in Tradefed refers to the common base class driving a suite execution. It's shared by all major test suites, specifically the Android Compatibility Test Suite (CTS) and Android Vendor Test Suite (VTS), and ensures a consistent execution experience across all suites.

We sometimes refer to ITestSuite as the suite runner.

The suite runner follows these steps when executing:

Load the module's configuration and determine which set should run.
Run each module:

Run module-level setup.
Run module tests.
Run module-level tear down.
Report the results.

Top-level setup
From a Tradefed point of view, ITestSuite is just another test. It's a complex one but is still just a test like any other IRemoteTest. So when specifying the suite runner in a Tradefed configuration, Tradefed follows the usual pattern of the configuration: running build_provider, target_preparer, test (our suite in this case), and target_cleaner.

This sequence in the Tradefed configuration containing the ITestSuite is the top-level setup.

Example:

```
<configuration description="Common config for Compatibility suites">

    <build_provider
class="com.android.compatibility.common.tradefed.build.CompatibilityBuildProvider" />
    <!-- Setup applied before the suite: so everything running in the suite will
    have this setup beforehand -->
    <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer">
        <option name="run-command" value="settings put global package_verifier_enable 0" />
        <option name="teardown-command" value="settings put global package_verifier_enable
1"/>
    </target_preparer>

    <!-- Our ITestSuite implementation -->
    <test
class="com.android.compatibility.common.tradefed.testtype.suite.CompatibilityTestSuite" />

    <result_reporter class="com.android.compatibility.common.tradefed.result.ConsoleReporter"
/>
</configuration>
```

Module metadata

We call module metadata extra information specified in the test module AndroidTest.xml. This metadata lets you specify additional information about the module, and modules can be filtered using the metadata.

Example metadata:

```
<option name="config-descriptor:metadata" key="component" value="framework" />
<option name="config-descriptor:metadata" key="parameter" value="instant_app" />
```

Example filter on metadata:

```
--module-metadata-include-filter component=framework
```

The above would run all the modules with a framework as component metadata.

Full AndroidTest.xml example:

```
<configuration description="Config for CTS Gesture test cases">
    <option name="test-suite-tag" value="cts" />
    <!-- Metadata -->
    <option name="config-descriptor:metadata" key="component" value="framework" />
    <option name="config-descriptor:metadata" key="parameter" value="instant_app" />
    <!-- End: metadata -->
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
```

```
        <option name="cleanup-apks" value="true" />
        <option name="test-file-name" value="CtsGestureTestCases.apk" />
    </target_preparer>
    <test class="com.android.tradefed.testtype.AndroidJUnitTest" >
        <option name="package" value="android.gesture.cts" />
        <option name="runtime-hint" value="10m50s" />
    </test>
</configuration>
Parameterized module
A special metadata type is parameter.


<option name="config-descriptor:metadata" key="parameter" value="instant_app" />
This metadata specifies that the module needs to be executed in a different mode, for example as an instant app, instead of a standard app mode.

All the possible modes or parameters are described by ModuleParameters and have an associated handler in ModuleParametersHelper that lets you change the module setup to execute in the particular mode.

For example, the instant app mode forces the APK installation as instant mode.

In order for the parameterization to occur, the command line needs to enable it with:


--enable-parameterized-modules
It's also possible to run a single given mode with:


--enable-parameterized-modules --module-parameter <Mode>

--enable-parameterized-modules --module-parameter INSTANT_APP
When a parameterized version of a module runs, it reports its results under a parameterized module name, for example CtsGestureTestCases[instant] versus base CtsGestureTestCases.
```

This entire explanation is about how **test suites** are structured and executed in **Tradefed (TF)** — the Android test harness that runs CTS, VTS, and other test suites.

Let's break it down **clearly and with examples** so it's no longer confusing.

🧩 What is a "Suite" in Tradefed?

A **suite** is a collection of test **modules** that are run together under a shared **top-level configuration** (setup, teardown, reporting, etc.).

Think of it like this:

> 🔧 **Suite = Test Runner + Setup + Many Modules (tests) + Reporting**

⚙️ Structure of a Suite

Here's the layered structure:

1. Suite-level (Top-level setup)

Common setup for all modules in the suite. Defined in a file like:

```
xml

<configuration description="Common config for Compatibility suites">

    <!-- Build Provider -->
    <build_provider
class="com.android.compatibility.common.tradefed.build.CompatibilityBuildProvider" />

    <!-- Setup before running any module -->
    <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer">
        <option name="run-command" value="settings put global package_verifier_enable 0" />
        <option name="teardown-command" value="settings put global package_verifier_enable 1" />
    </target_preparer>

    <!-- Suite runner -->
    <test
class="com.android.compatibility.common.tradefed.testtype.suite.CompatibilityTestSuite" />

    <!-- Results printed to console -->
    <result_reporter class="com.android.compatibility.common.tradefed.result.ConsoleReporter" />
</configuration>
```

> 🔁 This XML runs a **suite**, which internally finds and runs all modules defined in `AndroidTest.xml`.

---

2. Module-level Setup (AndroidTest.xml)

Each **test module** defines its own behavior and setup:

```
xml

<configuration description="Config for CTS Gesture test cases">
    <!-- Tag to group under CTS -->
    <option name="test-suite-tag" value="cts" />

    <!-- Module Metadata -->
    <option name="config-descriptor:metadata" key="component" value="framework" />
    <option name="config-descriptor:metadata" key="parameter" value="instant_app" />

    <!-- Setup before running this module -->
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="cleanup-apks" value="true" />
```

```
        <option name="test-file-name" value="CtsGestureTestCases.apk" />
    </target_preparer>

    <!-- Actual test execution -->
    <test class="com.android.tradefed.testtype.AndroidJUnitTest">
        <option name="package" value="android.gesture.cts" />
        <option name="runtime-hint" value="10m50s" />
    </test>
</configuration>
```

🔪 This defines how **CtsGestureTestCases.apk** should be installed, tested, and cleaned up.

🧠 Why we need all this?

Because Android testing must:

- Run across many devices
- Run many types of tests (unit, instrumentation, compatibility)
- Share setup/teardown logic
- Support retries and partial re-runs
- Handle **different device modes**, like normal vs. **instant apps**

This suite structure makes testing scalable, modular, and maintainable.

✅ Example Use Cases

1. Run only "framework" modules:

```
bash

run cts --module-metadata-include-filter component=framework
```

2. Enable parameterized testing (e.g., test as Instant App):

```
bash

run cts --enable-parameterized-modules --module-parameter INSTANT_APP
```

This will run test modules like:

- **`CtsGestureTestCases`** (normal)
- **`CtsGestureTestCases[instant]`** (instant app mode)

💡 Summary

Term	Meaning
Suite	Group of test modules with shared setup (via ITestSuite)
Module	Single test unit (apk + runner + teardown)
Top-level setup	Prepares the device for all modules
Module-level setup	Setup specific to that one test module
Metadata	Used to filter modules (e.g., by component or parameter)
Parameterized Module	Module that runs in a special mode (e.g., instant app)

Would you like a visual diagram to clarify this?

Pass options and filters to suite and modules

bookmark_border
First, ensure you understand Option Handling in Tradefed.

Suite setup describes the two layers that exist in the suite structure:

The top-level suite
The modules
In a non-suite Tradefed context, there is no need to think about it; every option goes to the full invocation. In a suite context, modules are kept isolated from the suite; so not all options are available at their level.

Pass options to the top-level suite
The top-level suite behaves like standard Tradefed configuration: the full configuration including the suite runner receives all the options like a non-suite Tradefed configuration.

Pass options to the modules
Modules by default do not receive any of the options passed to the command. They need to be explicitly targeted to receive the options through the module-arg option. This isolation of the modules options makes debugging easier.

Example:

cts-tradefed run cts --module-arg <module-name>:<option-name>:<option-value>

cts-tradefed run cts --module-arg CtsGestureTestCases:collect-tests-only:true
The syntax ensures that the targeted module will receive the given option.

There are additional ways to pass options to modules such as test-arg, which allows you to pass options to the test runner of each module based on the runner type or class.

Example:

cts-tradefed run cts --test-arg <test-class>:<option-name>:<option-value>

cts-tradefed run cts --test-arg com.android.tradefed.testtype.JarHosttest:collect-tests-only:true
The syntax does not target a particular module but rather all the test runners of the given class. test-arg considers only implementations of IRemoteTest as potential receiver of the options.

Pass options to a Java test class in a java_test_host
If you are adding an @Option to your java test class as part of a java_test_host build target you will need to use the following to inject that option:

cts-tradefed run cts --module-arg <module-name>:set-option:<option-name>:<option-value>
set-option in this context is the option of the HostTest runner from the test harness that wraps your java classes to execute them.

if your jar file target for the options contains multiple test classes, by default, they are all expected to have the @option specified or use the following syntax to target a single class:

cts-tradefed run cts --module-arg <module-name>:set-option:<class-name>:<option-name>:<option-value>
Pass filters to the suite
To filter some tests from a suite, we use --include-filter and --exclude-filter to respectively force include or exclude a particular test or module. Exclusion has priority.

They use this format: [abi] <module-name> [test name]

Examples:

--include-filter CtsGestureTestCases

--include-filter armeabi-v7a CtsGestureTestCases

--include-filter armeabi-v7a CtsGestureTestCases android.gesture.cts.GestureTest#testGetStrokes

Here's a simplified breakdown of **how to pass options and filters to a suite and its modules** in **Trade Federation (Tradefed)**:

✅ Context:

When running a **suite** (like CTS, GTS, etc.) via Tradefed (`cts-tradefed run cts`), there are **two layers**:

1. **Suite-level**: Global command
2. **Module-level**: Individual test modules (each defined by its own `AndroidTest.xml`)

🎯 1. Pass options to the suite-level (global)

These are options passed **normally**, like you would in standalone Tradefed.

```
bash

cts-tradefed run cts --shard-count 4 --retry 1
```

Everything here is received by the **suite runner** itself, not by individual modules inside.

🎯 2. Pass options to specific test modules

Modules are **isolated** and **don't receive global options** by default.
To send options **to modules**, use `--module-arg` or `--test-arg`.

🔹 a. `--module-arg`: Send options to a specific module

```
bash

cts-tradefed run cts \
  --module-arg CtsGestureTestCases:collect-tests-only:true
```

✅ Meaning: `collect-tests-only=true` will be passed to `CtsGestureTestCases`.

🔹 b. `--test-arg`: Send options to all tests of a specific class

```
bash

cts-tradefed run cts \
  --test-arg com.android.tradefed.testtype.JarHostTest:collect-tests-only:true
```

✅ Meaning: This will affect **any module** using `JarHostTest`.

🎯 3. Set option in `java_test_host` classes

If your test class has an `@Option`, you can inject it like this:

- **a. For all test classes:**

```
bash

cts-tradefed run cts \
  --module-arg CtsMyHostTest:set-option:my-option:true
```

- **b. Target a specific class:**

```
bash

cts-tradefed run cts \
  --module-arg CtsMyHostTest:set-option:com.my.MyClass:my-option:true
```

🎯 4. Apply filters to suite/modules

Use **`--include-filter`** and **`--exclude-filter`**.

- **Syntax:**

```
bash

--include-filter [abi] <module-name> [class#method]
```

- **Examples:**

```
bash

--include-filter CtsGestureTestCases
--include-filter armeabi-v7a CtsGestureTestCases
--include-filter armeabi-v7a CtsGestureTestCases android.gesture.cts.GestureTest#testGetStrokes
```

🟨 Exclusions always take **priority** over inclusions.

✅ Summary

Target	Use	Example
Suite	Normal options	**`--shard-count 4`**
Module	**`--module-arg`**	**`--module-arg CtsFoo:set-option:opt:true`**
Test class	**`--test-arg`**	**`--test-arg com.my.Test:option:true`**
Filters	**`--include-filter`**, **`--exclude-filter`**	**`--include-filter CtsFooClass#testBar`**

Would you like a working command example with real options or a visual flowchart for this?

Check system status

bookmark_border
System status checkers (SSCs) are defined at the suite-level configuration and run between each module. They perform checks to determine if the module changed and didn't restore some given states, for example changing a system property value.

SSCs are mainly used to ensure that module writers do not forget to clean up after their tests; but if they do, provide a trace of it so it can be addressed.

A secondary use is to also restore the original state when possible, for example dismissing the keyguard if it was left open.

System status checker XML definition

<system_checker class="com.android.tradefed.suite.checker.KeyguardStatusChecker" />
<system_checker class="com.android.tradefed.suite.checker.LeakedThreadStatusChecker" />
<system_checker class="com.android.tradefed.suite.checker.SystemServerStatusChecker" />
SSCs are defined under the system_checker tag in the Tradefed configuration XML.

Implementation
Every SSC must implement the ISystemStatusChecker interface, which provides the two main methods preExecutionCheck and postExecutionCheck that run before and after each module execution.

It's possible for a checker to implement only one of the two, or to implement both if there's a need to check the state before the module and compare it to the state after the module.

Several example implementations exist in Tradefed. Each implementation is recommended to focus on a single check to improve reusability. For example, SystemServerStatusCheck checks if the system_server process restarted on the device during the test suite execution. In the postExecutionCheck, it calls deviceSoftRestarted, which is defined in NativeDevice to check if the system_server process restarted.

Each operation returns StatusCheckerResult, which lets the harness decide if additional information, like a bug report, should be captured.

Where are they defined in CTS?
CTS system status checkers are defined in /test/suite_harness/tools/cts-tradefed/res/config/cts-system-checkers.xml.

How to find checker failures
By default, system checker failures show only in the logs and as bug reports captured for the

invocation with name following the format bugreport-checker-post-module-<module name>.zip.

This lets you find out after which module the bug report was generated.

It's possible to make the system checker report as a test failure itself by setting the --report-system-checkers option to true. This results in a test run showing as failed with the reason for failure being the status checker particular check.

In Tradefed (used in CTS/VTS testing), **System Status Checkers (SSCs)** are tools that **automatically check whether your device/system is still in a clean, valid state before and after each test module**. Their primary job is to **catch any side effects or system changes caused by a test module**, such as:

- Changed system properties
- Leftover background threads
- Crashed or restarted system services (like `system_server`)
- Unlocked screen left open after test (e.g. keyguard)

✅ Why do we need SSCs?

Tests are supposed to be isolated and **must not leave the system in a "dirty" or altered state**. If they do:

- It may affect other tests (flaky failures)
- Results may become unreliable
- Debugging becomes difficult

So SSCs help **automate cleanup detection** and optionally revert it, or fail the test if something went wrong.

🔧 Where are they defined?

They are declared in XML config files like:

```xml
<system_checker class="com.android.tradefed.suite.checker.KeyguardStatusChecker" />
<system_checker class="com.android.tradefed.suite.checker.SystemServerStatusChecker" />
```

These files are located here:

```
swift

/test/suite_harness/tools/cts-tradefed/res/config/cts-system-checkers.xml
```

🪛 How do they work?

Each SSC implements the interface:

```
java

ISystemStatusChecker
```

with these two key methods:

- **`preExecutionCheck()`** – Run **before** a test module
- **`postExecutionCheck()`** – Run **after** a test module

They can either:

- Just **log issues**
- Or **fail the test** if the issue is critical (if **`--report-system-checkers=true`** is set)

📦 Examples of Checkers

Checker Name	Purpose
`KeyguardStatusChecker`	Verifies the screen is locked/unlocked properly
`LeakedThreadStatusChecker`	Detects leftover Java threads from the module
`SystemServerStatusChecker`	Detects whether **`system_server`** has restarted during the test

🐛 Where to see checker failures?

- By default: only in logs and bugreports like:

  ```
  cpp

  bugreport-checker-post-module-CtsSomeTestCases.zip
  ```

- You can **force the failure to appear as a test failure** by:

  ```
  arduino

  --report-system-checkers true
  ```

✅ Example Scenario

Let's say you have a module **`CtsGestureTestCases`** that runs tests involving screen unlocking.

If the module forgets to lock the screen back, the **KeyguardStatusChecker** will catch it in **`postExecutionCheck()`**, log it, and capture a bugreport.

Summary

Feature	Description
Purpose	Detect system state pollution by test modules
Interface	**`ISystemStatusChecker`**
Methods	**`preExecutionCheck()`**, **`postExecutionCheck()`**
Result	Log or fail test, optionally capture bugreport
Config location	**`/test/suite_harness/tools/cts-tradefed/res/config/cts-system-checkers.xml`**
Common Checkers	**`Keyguard`**, **`SystemServer`**, **`LeakedThread`**, etc.

Would you like an example showing how a custom system status checker can be written in Java?

AndroidTest.xml structure

bookmark_border
The overall structure of the module configuration follows a similar pattern to the regular Tradefed XML configuration but with some restrictions due to the fact that they run as part of a suite.

List of allowed tags
AndroidTest.xml or more broadly module configuration can contain only the following XML tags: target_preparer, multi_target_preparer, test and metrics_collector.

Although that list looks restrictive, it allows you to precisely define test module setup needs and the test to run.

NOTE: See Tradefed XML configuration if you need a refresher on the different tags.

Objects such as build_provider or result_reporter will raise a ConfigurationException if attempted to be run from inside a module configuration. This is meant to avoid the expectation of these objects actually performing some task from within a module.

Example module configuration

```
<configuration description="Config for CTS Gesture test cases">
    <option name="test-suite-tag" value="cts" />
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="cleanup-apks" value="true" />
        <option name="test-file-name" value="CtsGestureTestCases.apk" />
    </target_preparer>
    <test class="com.android.tradefed.testtype.AndroidJUnitTest" >
        <option name="package" value="android.gesture.cts" />
        <option name="runtime-hint" value="10m50s" />
    </test>
</configuration>
```

This configuration describes a test that requires CtsGestureTestCases.apk to be installed and will run an instrumentation against the android.gesture.cts package.

Inclusion tags <include> and <template-include>
Using of <include> and <template-include> in module configs is discouraged. They are not guaranteed to work as expected.

Special case for metrics_collector tag
The metrics_collector is allowed but limited to the FilePullerLogCollector class in order to specify a given file or directory to be pulled and logged for the module. This is useful if you are leaving logs in a particular location and would like to automatically recover them.

Example configuration:

```
<configuration description="Config for CTS UI Rendering test cases">
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="cleanup-apks" value="true" />
        <option name="test-file-name" value="CtsUiRenderingTestCases.apk" />
    </target_preparer>
    <test class="com.android.tradefed.testtype.AndroidJUnitTest" >
        <option name="package" value="android.uirendering.cts" />
        <option name="runtime-hint" value="11m55s" />
        <option name="runner" value="android.uirendering.cts.runner.UiRenderingRunner" />
        <option name="isolated-storage" value="false" />
    </test>

    <!-- Collect the files in the dump directory for debugging -->
    <metrics_collector class="com.android.tradefed.device.metric.FilePullerLogCollector">
        <option name="directory-keys" value="/sdcard/UiRenderingCaptures" />
        <option name="collect-on-run-ended-only" value="true" />
    </metrics_collector>
</configuration>
```

What about build infos or downloads?

The definition of the allowed tags might give the incorrect impression that a module will not get any build information. This is not true.

The build information is provided from the suite-level setup and will be shared by all the modules of the suite. This allows a single top-level setup for the suite in order to run all the modules part of the suite.

For example, instead of each Compatibility Test Suite (CTS) module individually querying the device information, types, etc., the CTS suite-level setup (cts.xml) does it once and each module will receive that information if they request it.

In order for the objects in a module to receive the build information, they need to do the same as in regular Tradefed configuration: implement the IBuildReceiver interface to receive the IBuildInfo. See testing with device for more details.

Metadata fields
A large number of test modules include some metadata specifications, which each have a unique goal.

Example:

```
  <option name="config-descriptor:metadata" key="component" value="framework" />
  <option name="config-descriptor:metadata" key="parameter" value="instant_app" />
  <option name="config-descriptor:metadata" key="parameter" value="multi_abi" />
  <option name="config-descriptor:metadata" key="parameter" value="secondary_user" />
```

Component
The component metadata describes the general Android component the module intends to test. It doesn't have any direct impact on the test execution; it's primarily used for organizational purposes.

The up-to-date list of allowed components for CTS is available in CtsConfigLoadingTest. This test fails in presubmit if a non-existing component is added to a CTS module.

You can filter a suite run based on the components using module-metadata-include-filter and module-metadata-exclude-filter.

Example:

```
  --module-metadata-include-filter component framework
```

This example only runs the test module annotated with the framework component.

Parameter
The parameter metadata is informational and impacts the test execution. It specifies which Android mode the test module applies to. In this case, modes are limited to high level Android

modes, such as instant apps, secondary users or different abis.

During the suite run, if the mode applies to the test, several variation of the test module are created based on the mode. Each variation runs similar tests but under different modes.

instant_app: Create a variation of the tests that install APKs as instant apps.
multi_abi: Create a variation of the tests for each ABI supported by the device.
secondary_user: Create a variation of the tests that install APKs and run tests as a secondary user.
Metric collecting and post-processing for performance test modules
For performance test modules, module-level metrics_collector and metric_post_processor are allowed as they are essential to performance tests. The module-level metric collectors and post-processors can be module specific. It is not recommended to specify post-processors at both top-level and module-level.

A performance test module configuration must include the test-type metadata with value performance, like: xml <option name="config-descriptor:metadata" key="test-type" value="performance" /> Without this, if a test config includes metric_collector other than FilePullerLogCollector or any metric_post_processor, the test fails in presubmit.

Example performance test module configuration:

```
<configuration description="Runs sample performance test.">
    <!-- Declare as a performance test module -->
    <option name="config-descriptor:metadata" key="test-type" value="performance" />
    <option name="test-tag" value="hello-world-performance-test" />
    <test class="com.android.tradefed.testtype.HostTest" >
        <option name="class"
value="android.test.example.helloworldperformance.HelloWorldPerformanceTest" />
    </test>
    <!-- Add module-level post processor MetricFilePostProcessor -->
    <metric_post_processor
class="com.android.tradefed.postprocessor.MetricFilePostProcessor">
        <option name="aggregate-similar-tests" value="true" />
        <option name="enable-per-test-log" value="false" />
    </metric_post_processor>
</configuration>
```

Here's a breakdown of how to **write and structure an `AndroidTest.xml` file** for Trade Federation test modules — such as those used in CTS, VTS, or other Android test suites — with examples and explanations for each part.

✅ Purpose of `AndroidTest.xml`

The **`AndroidTest.xml`** file defines a **test module** that:

- Installs necessary APKs or files.
- Executes the test (on device or host).
- Optionally collects logs or metrics.

It's a simplified Tradefed XML config with strict rules due to suite integration.

🧱 Allowed Tags

In **`AndroidTest.xml`**, only the following tags are allowed:

Tag	Purpose
`<target_preparer>`	Prepares device: installs APKs, pushes files.
`<multi_target_preparer>`	Like **`target_preparer`**, but acts across multiple devices.
`<test>`	Defines the test to run (e.g., instrumentation, host test, shell test).
`<metrics_collector>`	Collects output files or logs from device.
`<metric_post_processor>`	Post-process metrics (only allowed in **performance** test modules).

📝 Example 1: Basic Instrumentation Test

```xml
<configuration description="Config for CTS Gesture test cases">
    <option name="test-suite-tag" value="cts" />

    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="cleanup-apks" value="true" />
        <option name="test-file-name" value="CtsGestureTestCases.apk" />
    </target_preparer>

    <test class="com.android.tradefed.testtype.AndroidJUnitTest">
        <option name="package" value="android.gesture.cts" />
        <option name="runtime-hint" value="10m50s" />
    </test>
</configuration>
```

🔍 Explanation:

- Installs **`CtsGestureTestCases.apk`**.
- Runs **`android.gesture.cts`** instrumentation package.
- 10-minute runtime hint.

📝 Example 2: With File Pulling Metrics

```xml
<configuration description="CTS UI Rendering test with file pull">
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="cleanup-apks" value="true" />
        <option name="test-file-name" value="CtsUiRenderingTestCases.apk" />
    </target_preparer>

    <test class="com.android.tradefed.testtype.AndroidJUnitTest">
        <option name="package" value="android.uirendering.cts" />
        <option name="runner" value="android.uirendering.cts.runner.UiRenderingRunner" />
        <option name="isolated-storage" value="false" />
    </test>

    <metrics_collector class="com.android.tradefed.device.metric.FilePullerLogCollector">
        <option name="directory-keys" value="/sdcard/UiRenderingCaptures" />
        <option name="collect-on-run-ended-only" value="true" />
    </metrics_collector>
</configuration>
```

📝 Example 3: Performance Test With Metric Post Processor

```xml
<configuration description="Runs HelloWorld performance test">
    <option name="config-descriptor:metadata" key="test-type" value="performance" />
    <option name="test-tag" value="hello-world-performance-test" />

    <test class="com.android.tradefed.testtype.HostTest">
        <option name="class"
value="android.test.example.helloworldperformance.HelloWorldPerformanceTest" />
    </test>

    <metric_post_processor class="com.android.tradefed.postprocessor.MetricFilePostProcessor">
        <option name="aggregate-similar-tests" value="true" />
        <option name="enable-per-test-log" value="false" />
    </metric_post_processor>
</configuration>
```

🎯 Metadata Options

Used to provide test info or modify behavior:

```xml
<!-- Required for filtering or variation creation -->
<option name="config-descriptor:metadata" key="component" value="framework" />
<option name="config-descriptor:metadata" key="parameter" value="multi_abi" />
```

Key Metadata:

- `component`: categorizes the test (e.g., framework, kernel, camera).
- `parameter`: triggers test **variations**:
  - `instant_app`: runs APKs as instant apps.
  - `multi_abi`: runs per supported ABI.
  - `secondary_user`: runs as secondary user.
- `test-type`: must be `performance` to allow metric processors.

🧠 Tips and Gotchas

- ❌ No `<build_provider>`, `<result_reporter>`, `<include>`, or `<template-include>` allowed.
- ✅ Metrics can only be collected via `FilePullerLogCollector` (unless marked as `performance`).
- ✅ Build info is passed **automatically** from the suite level — no need to define in module.
- ✅ Use `--module-metadata-include-filter` or `--module-arg` to control execution and filtering.

Would you like a template you can reuse or generate a specific XML config for your test module?

Configure sharding

bookmark_border
This page describes what is possible to tune for a suite module (AndroidTest.xml) via sharding and get the best speed performance during continuous execution in the lab. We will attempt to describe the options in a generic manner with the rationale for using each.

When running continuously a suite in the lab, the suite is usually sharded across several devices to reduce the overall completion time. The harness typically attempts to balance the execution time of each shard to minimize the overall completion time (when the last shard finishes); but due to the nature of some tests, we do not always have enough introspection and need the module owner to tune some behavior.

Shardable or not shardable?
It is possible to tag a module (AndroidTest.xml) with <option name="not-shardable" value="true" /> to notify the harness that it should not be sharded.

In a typical module, letting the harness shard your module (the default behavior) is the right thing to do. But in some cases, you might want to override that behavior:

When the setup of your module is expensive:
Sharding a module results in the preparation (install APK, push file, etc.) possibly run once per

device involved. If your module setup is long and expensive and not worth being replicated compared to the test's runtime, you should tag your module as not-shardable.

When the number of tests in your module is low:
Sharding a module results in all the test cases possibly executing independently on different devices. This relates to the first point; if your number of tests is low, you might end up with a single test or no test in some shards, which would make any preparation step quite expensive. Installing an APK for a single test case is usually not worth it, for example.

Instrumentation tests: Max number of shards?
An instrumentation test running through AndroidJUnitTest does not expose to the harness how many tests are part of the instrumentation until we actually install and run the APK. These operations are costly and cannot be executed at sharding time for all the modules part of the suite.

The harness might over-shard the instrumentation test and end up with some empty shards; sharding an instrumentation test with five tests in six shards results in five shards with one test and one shard with no tests. Each of these shards would require a costly APK installation.

So when the number of tests in the instrumentation test APK is low, tagging the module with <option name="not-shardable" value="true" /> would allow the harness to know sharding that module is not worth it.

The AndroidJUnitTest runner has a special option allowing it to specify the max number of shards it is allowed to shard into: <option name="ajur-max-shard" value="5" />.

This allows you to specify a maximum number of times the instrumentation can be sharded regardless of the number of shards requested at the invocation level. By default, the instrumentation will be sharded into the number of shards requested for the invocation.

For example, if your instrumentation test APK contains only two test cases but you still want to shard it, having a ajur-max-shard value of 2 would ensure you are not creating empty shards.

Sharding in the context of **CTS/VTS/Tradefed** is a powerful way to speed up test execution by **dividing tests into smaller chunks** (shards) that run **in parallel on multiple devices or threads**. Let's break this down clearly with **examples and rationale**.

🔹 What is Sharding?

Sharding = Splitting test cases to run in parallel

- If a module has 100 test cases and you use 5 shards → 20 tests per shard (approx).
- These shards can run **simultaneously on multiple devices**, drastically reducing total test time.

🔹 Why Sharding is Useful

✅ Benefits:

Scenario	Advantage
You have many tests	Run faster in parallel
You have many devices	Utilize them all efficiently
You want CI pipeline speed	Complete in minutes, not hours

🔹 When Not to Shard

❌ Set `not-shardable = true` when:

1. **Test setup is expensive**
   Example:
   - APK installation takes 1 minute
   - Test runtime is only 10 seconds
   - Sharding wastes more time than it saves

```xml
<option name="not-shardable" value="true" />
```

2. **Few tests in the module**
   - Example: Only 3 test cases → sharding into 4 devices = 1 shard is idle

---

🔹 Sharding for Instrumentation Tests

Problem:

The number of test cases in an **instrumentation APK is not known** until it's installed — this makes smart sharding hard.

🔧 Solution: `ajur-max-shard`

Example:

```xml
<option name="ajur-max-shard" value="2" />
```

This means:

- Even if harness asks for 10 shards,
- Your module will shard into **only 2 max**, avoiding over-sharding.

🔹 Visual Example

Let's say we have:

- Module with 10 tests
- 2 devices available

Case 1: Normal sharding

```
xml

<!-- Default: no not-shardable -->
```

Harness splits into 2 shards:

- Shard 0: 5 tests
- Shard 1: 5 tests

✅ Efficient

Case 2: Expensive setup, few tests

```
xml

<option name="not-shardable" value="true" />
```

Entire module runs on 1 device in sequence.

✅ Saves setup time

Case 3: Instrumentation test with 4 tests, over-sharded

```
xml

<option name="ajur-max-shard" value="2" />
```

Only 2 shards created:

- Each with 2 tests

- APK install done only twice, not 4 times

✅ Optimal balance

🔹 Summary Table

Option	Meaning	When to Use
`<option name="not-shardable" value="true" />`	Prevent sharding	Expensive setup or low test count
`<option name="ajur-max-shard" value="N" />`	Cap shards	Instrumentation tests with low test count

Would you like a real AndroidTest.xml example with these options in place?

Employ module controllers

bookmark_border
Each suite module (defined by AndroidTest.xml) can contain a special module_controller object that can alter some behavior of the module:

Whether to run the module or not based on some conditions
By implementing BaseModuleController and adding it to the AndroidTest.xml like this:

<object type="module_controller" class="com.android.tradefed.testtype.suite.module.<NAME>" />
The module controller will be used to determine whether the module should run or not, based on the public abstract RunStrategy shouldRun(IInvocationContext context); implementation.

Whether to collect some logs or not on failures
When running a full suite, it's possible to request at the suite level the collection of some logs on failures (screenshot, bugreport, logcat). But for some modules, a particular log requested might not have any value and will simply waste time to be collected. In that situation, a module can explicitly specify which logs they are interested in:

```
<object type="module_controller"
      class="com.android.tradefed.testtype.suite.module.TestFailureModuleController">
   <option name="screenshot-on-failure" value="<true OR false>" />
   <option name="bugreportz-on-failure" value="<true OR false>" />
   <option name="logcat-on-failure" value="<true OR false>" />
```

</object>
NOTE: Implementation of controllers should be generic if possible in order to maximize re-usability. And skipping a module based on its condition should be reviewed by the module owner to get the approval that skipping a module is the proper behavior for them.

Let's break this down with **examples** so you can understand how **module controllers** work in Trade Federation (Tradefed):

🔹 What is a Module Controller?

A **Module Controller** is a custom logic plug-in used in Tradefed to **conditionally control whether a test module should run or not** during a test invocation.

🔹 Use Case 1: Skip Module Based on Condition

Let's say you want a test to **run only on `arm64` devices**. You can write a custom controller like this:

1. Java: Create a Controller

```
java


package com.android.tradefed.testtype.suite.module;

import com.android.tradefed.testtype.suite.module.BaseModuleController;
import com.android.tradefed.invoker.IInvocationContext;

public class Arm64OnlyController extends BaseModuleController {
    @Override
    public RunStrategy shouldRun(IInvocationContext context) {
        String abi = context.getBuildInfos().get(0).getBuildAttributes().get("abi");
        if ("arm64-v8a".equals(abi)) {
            return RunStrategy.RUN;
        }
        return RunStrategy.SKIP;
    }
}
```

2. AndroidTest.xml: Register Controller

```
xml


<object type="module_controller"
        class="com.android.tradefed.testtype.suite.module.Arm64OnlyController" />
```

📌 What happens?

When the suite runs, the controller checks the ABI. If it's not `arm64-v8a`, it skips the module **cleanly**.

🔹 Use Case 2: Disable Log Collection on Failure (Save Time)

By default, test failures might collect:

- Screenshot
- Logcat
- Bugreport

But for some modules (like performance tests), you may not care about logs on failure. You can **disable log collection** like this:

AndroidTest.xml:

```
xml

<object type="module_controller"
        class="com.android.tradefed.testtype.suite.module.TestFailureModuleController">
    <option name="screenshot-on-failure" value="false" />
    <option name="bugreportz-on-failure" value="false" />
    <option name="logcat-on-failure" value="false" />
</object>
```

📌 What happens?
When any test in this module fails, Tradefed won't waste time collecting those logs.

🔹 RunStrategy Values:

Your controller returns one of the following:

- **`RUN`**: run the module.
- **`FULL_MODULE_BYPASS`**: skip the module entirely (not even attempted).
- **`SKIP`**: skip this module from test plan.

✅ Summary

Purpose	Method	XML Needed?	Example Class
Conditionally skip module	**`shouldRun()`**	✅	**`Arm64OnlyController`**
Skip log collection on failure	options via controller	✅	**`TestFailureModuleController`**

If you want, I can give you a **real example** of a controller class + test module XML together in a test suite.

Use suite retry

bookmark_border
A suite tends to include several test modules and can reach quite a large test corpus size. For example, the Android Compatibility Test Suite (CTS) includes hundreds of modules and hundreds of thousands test cases.

It becomes possible for a large amount of tests to fail due to poor isolation or devices going into a bad state.

The suite retry feature is meant to address those cases: It lets you retry the failures only instead of the full suites in order to rule out flakiness and poor isolation. If a test is consistently failing, the retry also fails; and you get a much stronger signal that there's a real issue.

Implement suite retry
The retry of results involves reading the previous results and rerunning the previous invocation.

The main interface driving the retry is ITestSuiteResultLoader, which lets you load a previous result, and the previous command line.

RetryRescheduler then uses this information to recreate the previous command and populate some filters in order to rerun only the previous failures or not executed tests.

Example suite retry: CTS
The retry configuration in CTS is:

```
<configuration description="Runs a retry of a previous CTS session.">
   <object type="previous_loader"
class="com.android.compatibility.common.tradefed.result.suite.PreviousResultLoader" />
   <test class="com.android.tradefed.testtype.suite.retry.RetryRescheduler" />

   <logger class="com.android.tradefed.log.FileLogger">
      <option name="log-level-display" value="WARN" />
   </logger>
</configuration>
```

This is applicable to most of the suites that extend it, for example VTS.

It would be invoked with:

```
cts-tradefed run retry --retry <session>
```

The session would be found by listing the previous results in the CTS console:

cts-tf > l r
Session Pass Fail Modules Complete Result Directory Test Plan Device serial(s) Build ID Product
0 2092 30 148 of 999 2018.10.29_14.12.57 cts [serial] P Pixel
The exact original command will be reloaded and rerun with extra filters. This means that if your original command included some options, they are also part of the retry.

For example:

cts-tradefed run cts-dev -m CtsGestureTestCases
The retry of the above is always bound to CtsGestureTestCases because we are retrying a command that involved only it.

Configure retry for CTS-style suite
In order for the retry to work, the previous results need to be exported in proto format. The following needs to be added:

<result_reporter class="com.android.compatibility.common.tradefed.result.suite.CompatibilityProtoResultReporter" />
This needs to be added to the XML configuration of the main command, and it results in a test-record.pb file being created in the result folder.

The CTS retry then loads data from a combination of the test-record.pb and the existing test_result.xml to prepare the retry invocation.

What is Suite Retry in Trade Federation?

Suite retry is a feature in Tradefed that allows you to **rerun only the failed or unexecuted tests** from a previous test session, instead of rerunning the entire test suite again.

This is especially useful in large test suites like:

- **CTS (Compatibility Test Suite)**
- **VTS (Vendor Test Suite)**

Where:

- A full test suite may include **hundreds of modules and thousands of test cases**
- A few test failures may be due to **flaky tests**, **temporary device issues**, or **poor test isolation**

❓ Why Do We Need Suite Retry?

Without suite retry:

- You'd need to rerun the **entire test suite** (hours of execution time) just to recheck a few failing tests.

With suite retry:

- You **only rerun failed/unexecuted tests**
- Saves **time**, **resources**, and **developer effort**
- Helps identify **true failures** vs **flaky issues**

✅ How Does Suite Retry Work?

1. **Run a normal test suite**, e.g.:

```
bash

cts-tradefed run cts
```

2. After execution, check results:

```
bash

cts-tf > l r
```

   This lists the test sessions with info like session ID, number of passed/failed modules, etc.

3. **Retry failed tests only**:

```
bash

cts-tradefed run retry --retry <session_id>
```

   Example:

```
bash

cts-tradefed run retry --retry 0
```

   This:

   - Loads the original command and filters from session 0
   - Reruns **only the failed/unexecuted tests** using `RetryRescheduler`

⚙️ Required Configuration

In your test XML config (**`compatibility-tradefed.xml`** or similar), add:

```xml
<configuration description="Runs a retry of a previous CTS session.">
    <object type="previous_loader"
            class="com.android.compatibility.common.tradefed.result.suite.PreviousResultLoader"
/>
    <test class="com.android.tradefed.testtype.suite.retry.RetryRescheduler" />

    <logger class="com.android.tradefed.log.FileLogger">
        <option name="log-level-display" value="WARN" />
    </logger>
</configuration>
```

Also, in the **original test run configuration**, you must enable **proto result export**:

```xml
<result_reporter
class="com.android.compatibility.common.tradefed.result.suite.CompatibilityProtoResultReporter"
/>
```

This writes a **`test-record.pb`** file needed by **`RetryRescheduler`**.

🧠 Example

You run this first:

```bash
cts-tradefed run cts-dev -m CtsGestureTestCases
```

Later, rerun the failures from the same session:

```bash
cts-tradefed run retry --retry 0
```

Tradefed uses the previous command line and failures from session 0 to rerun only the relevant tests.

✅ Benefits Summary

Feature	Benefit
Retry only failed/unexecuted tests	Saves time and resources
No need to know command-line flags again	Tradefed reuses original flags
Identifies flaky vs real failures	Stronger test signals
Available in CTS, VTS, etc.	Works for most suite-based tools

Would you like a **diagram** or **code walkthrough** of how `RetryRescheduler` works under the hood?

Multi-device modules

bookmark_border
This doc provides step-by-step instructions on how to create multi-device modules and calls out current limitations when known.

The sample
A CTS wifi-aware multi-device module is provided. It sends a message from one device over wifi and verifies the other device receives it.

The source for the module is at packages/modules/Wifi/tests/hostsidetests/multidevices/test/aware/.

We've annotated the example with as much comments as we feel are useful.

Step 1: Create the module folder
It is recommended to create a folder for your multi-devices module in the suite project it belongs to. For example: cts/hostsidetests/multidevices/. We recommend this so all multi-devices modules remain collocated at least at first which will make it easier to discover examples.

All files for this module should be put under their own module folder. For example: wifi_aware.

Step 2: Create the test
This is where you implement your test logic. It is highly dependent on what is being tested.

Create the Mobly test source, like: wifi_aware_test.py.

Step 3: Create the build file: Android.bp
Add an Android.bp file like packages/modules/Wifi/tests/hostsidetests/multidevices/test/Android.bp. Define a python_test_host module, similar to:

```
python_test_host {
    name: "CtsWifiAwareTestCases",
    main: "wifi_aware_test.py",
    srcs: ["wifi_aware_test.py"],
    test_suites: [
        "cts",
        "general-tests",
    ],
```

```
    test_options: {
        unit_test: false,
    },
    data: [
         // Package the snippet with the mobly test
        ":wifi_aware_snippet",
    ],
}
```

Specify the snippets for the test with the data field, which will be packed with the binary and can be located and installed in the test by ATest or in Continuous execution.

Mobly Bundled Snippets are available in Android at external/mobly-bundled-snippets/.

Optional: Create custom snippets
Some multi-device modules may require custom Mobly snippets. The sample test includes a wifi-aware snippet at packages/modules/Wifi/tests/hostsidetests/multidevices/com.google.snippet.wifi/aware/WifiAwareSnippet.java, which is built with Mobly Snippet Lib, available in Android at: external/mobly-snippet-lib/.

The snippet should be defined with android_test rule in Android.bp like standard instrumentation:

```
android_test {
    name: "wifi_aware_snippet",
    sdk_version: "current",
    srcs: [
        "CallbackUtils.java",
        "WifiAwareSnippet.java",
    ],
    manifest: "AndroidManifest.xml",
    static_libs: [
        "androidx.test.runner",
        "guava",
        "mobly-snippet-lib",
    ],
}
```

Step 4: Create the module config: AndroidTest.xml
Add an AndroidTest.xml file like packages/modules/Wifi/tests/hostsidetests/multidevices/test/aware/AndroidTest.xml. In this test configuration, you need to specify two devices for the test, similar to:

```
<configuration description="Config for CTS Wifi Aware test cases">
```

```
    <option name="test-suite-tag" value="cts" />
    <option name="config-descriptor:metadata" key="component" value="wifi" />
    <option name="config-descriptor:metadata" key="parameter" value="not_instant_app" />
    <option name="config-descriptor:metadata" key="parameter" value="not_multi_abi" />
    <option name="config-descriptor:metadata" key="parameter" value="not_secondary_user" />

    <device name="device1">
      <!-- For coverage to work, the APK should not be uninstalled until after coverage is pulled.
           So it's a lot easier to install APKs outside the python code.
      -->
      <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="test-file-name" value="wifi_aware_snippet.apk" />
      </target_preparer>
      <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer">
        <option name="run-command" value="input keyevent KEYCODE_WAKEUP" />
        <option name="run-command" value="wm dismiss-keyguard" />
      </target_preparer>
      <target_preparer class="com.android.tradefed.targetprep.PythonVirtualenvPreparer">
       <!-- Any python dependencies can be specified and will be installed with pip -->
       <option name="dep-module" value="mobly" />
      </target_preparer>
    </device>
    <device name="device2">
      <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="test-file-name" value="wifi_aware_snippet.apk" />
      </target_preparer>
      <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer">
        <option name="run-command" value="input keyevent KEYCODE_WAKEUP" />
        <option name="run-command" value="wm dismiss-keyguard" />
      </target_preparer>
    </device>

    <test class="com.android.tradefed.testtype.mobly.MoblyBinaryHostTest">
      <!-- The mobly-par-file-name should match the module name -->
      <option name="mobly-par-file-name" value="CtsWifiAwareTestCases" />
      <!-- Timeout limit in milliseconds for all test cases of the python binary -->
      <option name="mobly-test-timeout" value="60000" />
    </test>
</configuration>
```

Notce that:

This sample test has dependency on Mobly. Any dependency can be specified for PythonVirtualenvPreparer and will be installed with pip.
The mobly-par-file-namefor MoblyBinaryHostTest must match the module name as in Android.bp.
Do specify a mobly-test-timeout for the test. It is in milliseconds and applies to the complete

python binary execution (all test cases together). This is needed to avoid test cases hanging forever in case of some issues.
Each device tag can contain a distinct setup on each device, The Mobly config will receive them in the same order as specified in the XML.
Related to snippet apk installation:

The initial POC has been updated to install snippet apks via target_preparer due to conversation with Coverage team: In order to ensure coverage measurements are not deleted too early, uninstalling by Harness rather than by test code in Python binaries offer better guarantees in terms of timing.
Step 5: Run test locally: atest
Currently, multi-device tests only runs on physical devices. Before running the test, verify that your test devices are in proper state. The command adb devices should report the list of your connected devices. If the list contains devices not intended for testing, specify the devices for the test using -s flag.

For wifi tests, make sure wifi is enabled for the devices (after factory reset).

You can run the test locally with atest:

$ atest CtsWifiAwareTestCases
You should see the number of devices used in the summary heading in atest output, something like Test executed with 2 device(s).

Troubleshooting
If the test fails when running locally due to:

Virtualenv error

java.io.IOException: Cannot run program
"virtualenv": error=2, No such file or directory
Please ensure virtualenv is in your PATH. Adding "~/.local/bin" to PATH should fix it. if virtualenv isn't installed follow: https://virtualenv.pypa.io/en/latest/installation.html

Expected to get at least 2 controller objects, got 1
Test modules are either multi-devices or single-device, there are no mixed-modules. If you attempt to run a multi-devices module without multiple devices you will see this error:

Expected to get at least 2 controller objects, got 1

Executing the module in multi-devices mode will solve the issue.

For CTS: You can use sharding to trigger it (For example: --shard-count 2) or run cts-multidevces.

📱 What Are Multi-Device Modules in Trade Federation?

Multi-device modules are test modules that require more than one device to execute test logic—e.g., sending a message from one device and receiving it on another over Wi-Fi. They are used to validate features that span multiple physical Android devices.

✅ Why Use Multi-Device Modules?

Some use cases require **interaction between devices**, such as:

- Wi-Fi Aware (NAN) communication between two phones
- Bluetooth pairing and data transfer
- Casting/mirroring from one phone to a display device
- Nearby device discovery and sync

Single-device test logic can't validate these cross-device scenarios. That's where multi-device test modules come in.

🧱 Key Components Overview

Here's a breakdown of the core steps and components involved:

🗂️ Step 1: Create the Module Folder

Organize tests in a logical directory, like:

```
swift

cts/hostsidetests/multidevices/wifi_aware/
```

🔪 Step 2: Create the Test Logic (Mobly Test)

Write your Mobly-based Python test, e.g., **`wifi_aware_test.py`**, that includes test cases with logic for:

- One device sending messages
- The second device receiving and validating them

🛠️ Step 3: Build File – Android.bp

Define a **`python_test_host`** module to package the Mobly test:

```
python

python_test_host {
    name: "CtsWifiAwareTestCases",
    main: "wifi_aware_test.py",
    srcs: ["wifi_aware_test.py"],
    test_suites: ["cts", "general-tests"],
    test_options: {
        unit_test: false,
    },
    data: [
        ":wifi_aware_snippet",
    ],
}
```

This bundles the Python test and snippet APK together.

📲 Optional: Custom Snippets (Java Code on Device)

Mobly test might interact with Android services (e.g., Wi-Fi Aware APIs). These are implemented in **Mobly Snippets** in Java (like **`WifiAwareSnippet.java`**).

Define and build the snippet like:

```
python

android_test {
    name: "wifi_aware_snippet",
    sdk_version: "current",
    srcs: [
        "CallbackUtils.java",
        "WifiAwareSnippet.java",
    ],
    manifest: "AndroidManifest.xml",
    static_libs: [
        "mobly-snippet-lib",
        "androidx.test.runner",
        "guava",
    ],
}
```

⚙️ Step 4: Configuration File – AndroidTest.xml

This defines:

- Number of devices
- Snippet APK installation
- Environment setup
- Mobly runner

```
xml

<device name="device1">
    <!-- Install the snippet APK -->
    <target_preparer class="com.android.tradefed.targetprep.suite.SuiteApkInstaller">
        <option name="test-file-name" value="wifi_aware_snippet.apk" />
    </target_preparer>
    <!-- Wake up the device -->
    <target_preparer class="com.android.tradefed.targetprep.RunCommandTargetPreparer">
        <option name="run-command" value="input keyevent KEYCODE_WAKEUP" />
        <option name="run-command" value="wm dismiss-keyguard" />
    </target_preparer>
    <!-- Install python dependencies -->
    <target_preparer class="com.android.tradefed.targetprep.PythonVirtualenvPreparer">
        <option name="dep-module" value="mobly" />
    </target_preparer>
</device>

<device name="device2">
    <!-- Same setup for second device -->
</device>

<test class="com.android.tradefed.testtype.mobly.MoblyBinaryHostTest">
    <option name="mobly-par-file-name" value="CtsWifiAwareTestCases" />
    <option name="mobly-test-timeout" value="60000" />
</test>
```

Notes:

- Device setup is **duplicated for each device**
- **`mobly-test-timeout`** ensures tests don't hang forever
- The snippet is **installed outside** the Python test to avoid premature uninstall (important for code coverage)

▶️ Step 5: Running the Test

Use **`atest`**:

```
bash

atest CtsWifiAwareTestCases
```

Make sure:

- You have **at least 2 physical devices** connected
- Devices are **awake**, unlocked, and **Wi-Fi is enabled**
- You see output like: **`Test executed with 2 device(s).`**

🪛 Troubleshooting Tips

Error	Fix
`Expected to get at least 2 controller objects, got 1`	Ensure 2 devices are connected and visible via **`adb devices`**
`Cannot run program "virtualenv"`	Install virtualenv and ensure it's in **`PATH`**
Snippet not working	Check snippet APK is built and properly installed via target preparer
Mobly test not running	Ensure **`mobly-par-file-name`** matches test name in Android.bp

✅ Summary

Component	Purpose
`wifi_aware_test.py`	Main Python Mobly test logic
`wifi_aware_snippet`	Java snippet that runs on device for API access
`Android.bp`	Defines test module and snippet
`AndroidTest.xml`	Sets up devices and test execution
`atest`	Tool to run tests locally

Would you like a **template** or a **simplified test case example** to start from?

Develop Tradefed

bookmark_border
This section is aimed at Tradefed developers. If you are interested in extending TF or adding new test support, this is the section for you.

Open source
If you plan to use the open source variant of Trade Federation, use these commands to check out and build the AOSP android-latest-release branch of Trade Federation:

cd <sourceroot>
mkdir android-latest-release
cd android-latest-release
repo init -u https://android.googlesource.com/platform/manifest -b android-latest-release
repo sync -c -j8
source build/envsetup.sh
lunch aosp_cf_arm64_only_phone-userdebug # or any other device target
m -j tradefed-all
See Development Environment for more details.

All open-sourceable Trade Federation code is stored in the tools/tradefederation/ git project of AOSP. Please keep the open source guidelines in mind when writing code and submitting changes.

Alternatively, if you are creating tests/utilities that use Trade Federation but aren't working on the framework itself, consider placing your work inside one of the Trade Federation contrib projects to speed approvals.

Coding style
Trade Federation follows the Android coding style guidelines, with the following clarifications: interface names are prefixed with 'I' e.g. ITestDevice.

Develop using Eclipse
If you are interested in using Eclipse in developing Tradefed, refer to Set up Eclipse IDE for tips on setting up your environment.

Run Tradefed's tests
You made a change to Trade Federation and you are searching how to test it? see Running Trade Federation's tests.

Architecture
If you seek a deeper understanding of the innerworkings of Tradefed, see the Architecture section.

Here's a **simple explanation** of the section you posted about **developing Tradefed**:

🛠️ What is Tradefed development?

This guide is for **developers** who want to:

- Modify **Trade Federation (Tradefed)** itself.
- Add **new features** or support **new types of tests**.
- Contribute to **Android Open Source Project (AOSP)** Tradefed code.

📥 How to get the Tradefed source code (open-source version)

You can download the latest Tradefed source like this:

```
bash

cd <sourceroot>
mkdir android-latest-release
cd android-latest-release

# Initialize repo for Android source (latest release)
```

```
repo init -u https://android.googlesource.com/platform/manifest -b android-latest-release

# Sync source files (use multiple threads for speed)
repo sync -c -j8

# Set up the build environment
source build/envsetup.sh

# Choose a target device (ex: AOSP emulator)
lunch aosp_cf_arm64_only_phone-userdebug

# Build Tradefed and its dependencies
m -j tradefed-all
```

This downloads and builds the **Tradefed** system from source.

📂 Where is Tradefed source code located?

In the AOSP source tree:

```
bash

tools/tradefederation/
```

If you plan to contribute code, follow **open-source coding rules**.

👩‍💻 Writing code that uses Tradefed (but not modifying it)?

Put your work in **Tradefed contrib projects**, like:

```
bash

tools/tradefederation/contrib/
```

This helps get your code reviewed and merged faster.

✏️ Coding Style:

- Follows Android coding style.
- Interfaces must start with `I` (like `ITestDevice`).

🧪 How to test your changes to Tradefed

You can run Tradefed's own test cases to make sure your changes are working properly. See the section: **Running Trade Federation's tests** (link in docs).

🧠 Want to understand how Tradefed works inside?

Check the **Architecture** section to learn:

- How Tradefed components interact
- How tests are scheduled and run
- Internals like listeners, test runners, devices, etc.

🖥️ Using Eclipse for development?

You can use Eclipse IDE to write and debug Tradefed code. There is a setup guide in the docs.

Would you like me to:

- Explain how to write a custom Tradefed test?
- Help you modify Tradefed itself?
- Show a small example module?

Let me know your goal!

> Test harness guidelines
>
> bookmark_border
> This section provides guidance for various aspects of test harness development and testing.
>
> Test expectations
> Any new code is expected to have appropriate testing; see include unit and functional tests on how this is done for Trade Federation.
>
> Feature flags
> Trade Federation is used to drive tests across Android, and bugs can have a large impact. We recommend any new feature be disabled by default and under a feature flag.
>
> Feature flags in TF rely on our @Option support to enable them via command line or configuration. See Option Handling for details about our options.
>
> Non-core contribution guidelines
> See the guidelines for non-core contributions.

Here's a **simple explanation** of the **Test Harness Guidelines** section in the context of Trade Federation (Tradefed):

✅ Test Expectations

Whenever you add **new code** to Tradefed:

- You **must write tests** for it.
- These tests should include:
  - **Unit tests** → test small parts (methods, classes).
  - **Functional tests** → test features end-to-end.

📌 *Why?*
Because Tradefed runs tests for the entire Android ecosystem, bugs can break many systems. So code must be reliable.

You can refer to the AOSP guidelines on **how to write and run these tests.**

🧪 Feature Flags

If you're adding a **new feature**, **don't enable it by default.**

Instead:

- Hide it behind a **feature flag** (like a switch).
- This prevents accidental breakage while the feature is still experimental.

🔧 Feature flags in Tradefed use `@Option` annotations:

These allow you to **turn on/off features** via:

- Command line:

```
bash

tradefed.sh run my-test --enable-feature-x true
```

- XML configuration.

📌 See: **Option Handling** in Tradefed documentation for how to define and use `@Option`.

📤 Non-core Contributions

If your code **does not touch Tradefed core** (like utilities, helpers, plugins), follow the **non-core contribution guidelines**.

Examples:

- Adding your own test runner
- Adding utility classes for device management
- Making a test config

This ensures faster review and avoids affecting core behavior.

Summary

Topic	What to do
Test Expectations	Write **unit** and **functional** tests for new code
Feature Flags	Disable new features by default, control with **`@Option`**
Non-core Contributions	Follow special guidelines if you're not changing core Tradefed logic

---

Would you like a **code example** for:

- Writing a feature with **`@Option`**?
- Writing a unit test in Tradefed?
  Let me know!

Tradefed architecture

bookmark_border
This section describes the internals of Tradefed and their relationships. See the linked subpages for more details.

Test configurations
Test configurations in Tradefed are described in an XML format. Understanding the structure of the configuration is key to running and customizing tests.

Structure of TF configurations
High-level structure
Templates and includes
Configuration object
Global TF configurations
A Global Configuration file is a special Tradefed XML configuration that is loaded when Tradefed starts via the TF_GLOBAL_CONFIG environment variable. It loads objects related to the Tradefed

instance scope that will affect the overall harness behavior.

Global Configuration details

Keystore
Keystore allows injection of command line options to Tradefed coming from a keystore in order to avoid referencing the value directly on the command line. This can be used to hide passwords from the command line by retrieve passwords from the keystore directly.

Keystore details

Device manager
The device manager is responsible for keeping track of the state of devices on a running instance of Tradefed. Aspects such as allocation status and online status are monitored.

Device states
Device allocation
Device detection sequence
Test command scheduler
The test command scheduler in Tradefed takes commands to run, associates them with devices, and starts a test invocation.

Test command life cycle
Build provider
Build provider is the first step of any test invocation. It downloads resources needed to set up and run the tests (build images, test APKs, and more.). It also references them in a BuildInfo object that will be passed to the test. Locally available resources can also be linked in the BuildInfo object.

Build provider details
Build info details
Target preparer and cleaner
Target preparer offers optional actions that can be taken to configure the target under test into a certain state, for example flashing the device, setting certain properties, and connecting to Wi-Fi.

Add a new target preparer/cleaner
Add a new multi target preparer
Test runner
A test runner in Tradefed refers to the object responsible for the actual test execution. Different test runners drive test execution in different ways; for example, an instrumentation test runner will be very different from a JUnit test runner.

Test runner structure
Result reporter
Result reporter in Tradefed refers to the object that will send the results to a particular

destination. Each implementation is usually specialized for different result back-ends. And the result reporter is in charge of converting the Tradefed results format into the destination format.

This flexible design allows any test to report to any of the results destinations and to easily have more tests added in an isolated way.

Add a result reporter
Result reporter and logs
Result reporter summary
Metrics collector
Metrics collector is a special object in Tradefed, orthogonal to the test execution. It allows collection of information at different points of the test lifecycle (for example, test start, test end). Since the collector is decoupled from the test itself, the points can be swapped, added, and removed without having to change the test itself.

Host-driven metrics collectors
Device side metrics collectors
Host-wide setup
This section describes setups that are applicable to a full Tradefed instance's running. These options affect the behavior of the harness as a whole in order to adapt to different environments, for example being in a restricted network.

Global Configuration
Host options
Additional features
The following sections describe general usage of Tradefed rather than Tradefed objects.

Tradefed sharding
When the test corpus is large or takes a long time to execute, it's possible to split it across several devices. We refer to this split as sharding. This section describes how sharding works and how it is configured.

Sharding details

Using SL4A
Tradefed supports the scripting layer for Android, SL4A; this is an automation toolset for calling Android APIs in a platform-independent manner.

SL4A with Tradefed details

Dynamic @option download
In some cases, the files needed by a test or some particular operation are not available locally. This feature allows Tradefed to get these files from a remote location without going through a build provider.

> Dynamic @option download

Here's a **simplified overview of the Trade Federation (Tradefed) architecture**, covering each component mentioned in your reference:

🔧 1. Test Configurations (XML-based)

Tradefed tests are defined using **XML configuration files**, which describe:

- What tests to run
- On which device
- With what setup

These XMLs include:

- `build_provider`
- `target_preparer`
- `test_runner`
- `result_reporter`

➡️ **Templates and includes** allow you to reuse config pieces across test definitions.

🌍 2. Global Configuration

Set via the `TF_GLOBAL_CONFIG` environment variable, this config is loaded **at startup**, affecting the whole TF process:

- Device manager settings
- Logging setup
- Default key store location

🔐 3. Keystore Support

Use keystore to **hide sensitive values**, like passwords, from the command line.

- Passwords are retrieved dynamically from the keystore.
- Reduces accidental exposure in logs or shell history.

📱 4. Device Manager

Responsible for managing all devices connected to the Tradefed host:

- Tracks **online/offline status**
- Handles **device allocation**
- Manages **device detection and recovery**

📦 5. Build Provider

The first step of a test invocation:

- **Downloads or fetches** the build artifacts (system images, test APKs).
- Packs them into a `BuildInfo` object.

🔗 BuildInfo is passed throughout the invocation.

⚙️ 6. Target Preparer & Cleaner

- **Preparer**: Sets up the device before tests. E.g., flash firmware, push files, change settings.
- **Cleaner**: Cleans up after test. E.g., wipe data or restore factory state.

There are **single-target** and **multi-target** versions.

🧪 7. Test Runner

Responsible for **running the test logic**.

- Examples: `InstrumentationTest`, `GTest`, `JUnitTest`, etc.
- Each runner has its own logic (e.g., ADB shell, instrumentation, JUnit engine).

📢 8. Result Reporter

Sends the results to a desired destination:

- Log file
- Test dashboard (e.g., ResultDB, TFC)
- Email, bug tracking system, etc.

Flexible format: allows plugging in custom reporters.

📊 9. Metrics Collector

Collects performance and usage data **during test lifecycle**:

- Before test run
- After test run
- Per test case

Types:

- **Host-based collectors**
- **Device-side collectors**

🧭 10. Test Command Scheduler

Orchestrates **which test runs on which device**:

- Picks the right device
- Launches test invocation
- Manages concurrency and queueing

🧩 11. Sharding

Allows **splitting large test suites across devices** to speed up execution.

- Static: You manually split tests.
- Dynamic: TF splits automatically.

🎌 12. Dynamic @Option Download

When a file or resource is not present locally, TF can **download it from a remote source** using a special `@Option`.

🔪 13. InstrumentationTest & ITestCollector

- `InstrumentationTest` implements `ITestCollector`
- Supports `--collect-tests-only` mode:
  - Runs setup
  - **Does not run tests**, just lists them
  - Useful for dry-runs and validation

🧠 Summary Diagram:

```
plaintext

             [ XML Test Configuration ]
                        |
      ------------------------------------------------
     |                  |                   |           |
[Build Provider]  [Target Preparer]  [Test Runner]  [Result Reporter]
                        |                   |           |
                 [Device Manager]   [Metrics Collector]
                        |
             [Test Command Scheduler]
```

Let me know if you want a **code example**, **custom test config**, or **walkthrough of a dry run (`--collect-tests-only`)**.

> yes in simple words

Let's break down the **Tradefed (Trade Federation)** architecture in **simple words** with **real-world examples** to help you understand better:

🔧 What is Tradefed?

Tradefed is a **test harness**—basically, a program that **runs tests on Android devices** automatically. It's mainly used by Google and OEMs to test Android system components.

🧱 Tradefed Architecture (Simplified)

1. Test Configuration (XML)

- Think of this like a **recipe** or **instruction sheet**.
- You write what to test, how to prepare the device, where to get the app or image, how to report results, etc.
- Example:

```xml

<configuration>
    <test class="com.android.tradefed.testtype.InstrumentationTest" />
</configuration>
```

---

2. Global Configuration

- This is like a **startup configuration** for the whole Tradefed tool.
- It's set using an environment variable: `TF_GLOBAL_CONFIG`
- It controls how Tradefed behaves as a whole.
- Example: Where logs are saved, how devices are detected, how credentials are stored.

---

3. Keystore

- Used to **securely store sensitive data** (like passwords or API keys).
- Instead of writing a password in your XML, you fetch it from the keystore.

---

4. Device Manager

- Keeps track of all connected Android devices.
- Knows which devices are **free, busy, offline, or allocated** for tests.

---

5. Test Command Scheduler

- Think of this like a **traffic cop**.
- It assigns available devices to the test commands you give.

---

6. Build Provider

- Provides the **test files or Android builds** needed for the test.
- For example, it can fetch a build from a server or use a local file.
- Example: Pulls the latest Android image to flash on the device before testing.

---

7. Target Preparer / Cleaner

- These are like **setup and cleanup workers**.
- Preparer: Sets up the device (e.g., flash a ROM, connect Wi-Fi).
- Cleaner: Restores or resets the device after tests.

---

8. Test Runner

- This is the **main test executor**.

- Runs the actual test like:
  - Instrumentation tests
  - JUnit tests
  - GTest for native code
- Each type of test has its own runner.

---

9. Result Reporter

- This collects and **sends your test results** somewhere.
- Could be:
  - Console output
  - File
  - Server (like a dashboard)

---

10. Metrics Collector

- These are tools that gather **extra info during testing** like:
  - CPU usage
  - Battery level
  - App launch time

🧩 Additional Features

🧵 Sharding

- Run tests in **parallel across multiple devices**.
- Helps when tests take too long.
- Like splitting 100 questions among 4 people instead of 1.

📜 SL4A Support

- Lets you use **scripting (Python, etc.)** to run Android API commands easily.

🌐 Dynamic @Option Download

- Downloads missing files (like configs or test data) **on the fly** from a remote server.

🧪 Types of Tests You Can Run

- **CTS** (Compatibility Test Suite)
- **VTS** (Vendor Test Suite)
- **Instrumentation tests** (AndroidJUnitRunner)
- **Native tests** (GTests)
- **UI tests** (via uiautomator)
- **Custom tests** (your own logic)

🏁 Summary

Component	What It Does
Configuration	Describes what & how to run tests
Global Config	Startup settings for Tradefed
Device Manager	Tracks all Android devices
Scheduler	Assigns test jobs to devices
Build Provider	Gets build files/images
Target Preparer	Sets up test device
Test Runner	Executes the actual test
Result Reporter	Logs and reports test results
Metrics Collector	Gathers performance/statistics
Sharding	Speeds up tests by using multiple devices

Would you like to see a **real XML configuration example** to understand how all these are written?

> types of test and their test runner

Great! Let's go over the **types of tests Tradefed can run** and the **test runners** used for each type, in **simple language with examples**:

🔍 Types of Tests in Tradefed & Their Test Runners

Test Type	Used For	Test Runner Class
Instrumentation Test	Testing Android apps via AndroidJUnitRunner	**`com.android.tradefed.testtype.InstrumentationTest`**
GTest (Native Test)	Native C/C++ code testing (Google Test framework)	**`com.android.tradefed.testtype.GTest`**
Host Test (JUnit)	Java JUnit tests that run **on the host** machine	**`com.android.tradefed.testtype.HostTest`**
UI Automator Test	Testing UI flows across apps	**`com.android.tradefed.testtype.UiAutomatorTest`**
VTS Test (HAL)	Vendor HAL interface tests (HIDL/AIDL)	**`com.android.tradefed.testtype.VtsMultiDeviceTest`**
CTS Test	Compatibility Test Suite (for Android compliance)	**`com.android.compatibility.common.tradefed.testtype.CTS*`**
Mobly Test	Python-based test framework for device stacks	**`com.android.tradefed.testtype.mobly.MoblyBinaryHostTest`**
Python Binary Test	Running Python test scripts	**`com.android.tradefed.testtype.python.PythonBinaryHostTest`**
Shell Test	Shell script test on device	**`com.android.tradefed.testtype.shell.AndroidJUnitTest`**
Test Suite (SuiteRunner)	Whole test suites like CTS, VTS, GTS	**`com.android.tradefed.testtype.suite.SuiteTestFilter`** + **`SuiteTestRunner`**

💡 Examples by Type

1. InstrumentationTest

- Runs Android instrumentation via **`adb shell am instrument`**.
- Example:

```xml
<test class="com.android.tradefed.testtype.InstrumentationTest">
  <option name="package" value="com.example.myapp.test" />
</test>
```

2. GTest

- Native C++ test.
- Example:

```xml
<test class="com.android.tradefed.testtype.GTest">
  <option name="module-name" value="libmytest" />
</test>
```

3. HostTest

- Runs JUnit on host.
- Example:

```xml
<test class="com.android.tradefed.testtype.HostTest">
  <option name="class" value="com.example.MyHostUnitTest" />
</test>
```

4. VtsMultiDeviceTest

- Runs tests defined in **`VTS`** using shell/python/native code.
- Used with HALs or vendor components.
- Test cases stored in test plans.

5. CTS

- Uses its own test runners, like:
  - **`CtsTest`** (normal modules)
  - **`CtsAbiTest`** (ABI-specific modules)

🔪 Where Test Types Run

Test Type	Runs On
InstrumentationTest	Android device
GTest	Android device
HostTest	Host machine
Mobly/Python	Host, talks to device
ShellTest	Android device
VtsMultiDeviceTest	Host + device

Test Type	Runs On
CTS/VTS	Both

Would you like a visual diagram of how test runners connect to devices and test types?

High-level structure of Tradefed XML configuration

bookmark_border
Tradefed's configurations follow an XML structure to describe the test to be run and the preparation/setup steps to be done.

In theory, everything can be defined in the XML for a single command. But in practice, it is more practical to have base template XML files and customize them with extra command line parameters.

Structure

```
<configuration description="<description of the configuration>">
    <!-- A build provider that takes local device information -->
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

    <!-- Some target preparation, disabled by default -->
    <target_preparer class="com.android.tradefed.targetprep.PreloadedClassesPreparer">
        <option name="disable" value="true" />
    </target_preparer>

    <!-- One test running some unit tests -->
    <test class="com.android.tradefed.testtype.HostTest">
        <option name="class" value="com.android.tradefed.build.BuildInfoTest" />
    </test>

    <!-- [OPTIONAL] -->
    <logger class="com.android.tradefed.log.FileLogger">
        <option name="log-level" value="VERBOSE" />
        <option name="log-level-display" value="VERBOSE" />
    </logger>

    <!-- [OPTIONAL] -->
    <log_saver class="com.android.tradefed.result.FileSystemLogSaver" />

    <!-- As many reporters as we want -->
    <result_reporter class="com.android.tradefed.result.ConsoleResultReporter" />
```

```
    <result_reporter class="com.android.tradefed.result.suite.SuiteResultReporter" />
    <result_reporter class="com.android.tradefed.result.MetricsXMLResultReporter"/>
</configuration>
```

The overall Tradefed XML is delimited by <configuration> tags. Tradefed objects are defined in their own tags, for example: build_provider, target_preparer, test, etc. Their individual purposes are described in more detail in the Architecture section.

Each object has the Java class associated with the object defined in class= that is resolved at runtime; so as long as the JAR file containing the class is on the Tradefed Java classpath when running, it will be found and resolved.

Orders of Tradefed objects
The order of the different tags does not matter. For example, it makes no difference if build_provider is specified after target_preparer. The flow of the test invocation is enforced by the harness itself, so it will always call them in the right order.

The order of objects with the same tag does matter. For example, two target_preparer objects defined will be called in their order of definition in the XML. It is important to understand this as it can change the end state of the device setup. For example, flashing then installing an apk would not be the same as installing an apk and flashing since flashing would wipe the device.

Here's a **simple explanation** of the **high-level structure** of a **Tradefed XML configuration file**:

💡 What is it?

A **Tradefed XML file** tells Tradefed:

- **What to test**
- **How to prepare the device**
- **Where to get the build**
- **How to log/report the results**

📦 Main Parts of the XML:

```xml
<configuration description="...">
    <!-- 1. Build provider -->
    <build_provider class="..."/>

    <!-- 2. Target preparation -->
    <target_preparer class="...">
        <option name="..." value="..." />
    </target_preparer>

    <!-- 3. The test itself -->
```

```
    <test class="...">
        <option name="class" value="..." />
    </test>

    <!-- 4. [Optional] Logging -->
    <logger class="...">
        <option name="..." value="..." />
    </logger>

    <!-- 5. [Optional] Log saving -->
    <log_saver class="..."/>

    <!-- 6. [Optional] Result reporters -->
    <result_reporter class="..."/>
</configuration>
```

🧩 What Each Tag Means:

Tag	Purpose
<build_provider>	How/where to get the Android build (e.g., use the already running device)
<target_preparer>	Setup steps for the device (e.g., install APKs, wipe data, push files)
<test>	Defines the test to run (e.g., unit test, instrumentation test)
<logger>	Controls how verbose the logs are
<log_saver>	Where logs are saved (e.g., file system)
<result_reporter>	Controls how test results are shown/saved (e.g., console, XML)

✅ Important Notes:

- Tags like **<build_provider>**, **<test>** are **essential**.
- Tags like **<logger>**, **<log_saver>** and **<result_reporter>** are **optional** but useful.
- The **order of different tag types** (e.g., test vs. build_provider) **does not matter**.
- But if you have **multiple of the same tag** (e.g., multiple **<target_preparer>**), **order matters**.

🔪 Example Explained:

```
xml

<configuration description="Run unit test example">

    <!-- Use current device as the source -->
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

    <!-- Disable a target setup step -->
    <target_preparer class="com.android.tradefed.targetprep.PreloadedClassesPreparer">
        <option name="disable" value="true" />
    </target_preparer>

    <!-- Run unit test on the host machine -->
```

```
    <test class="com.android.tradefed.testtype.HostTest">
        <option name="class" value="com.android.tradefed.build.BuildInfoTest" />
    </test>

    <!-- Save logs and show detailed output -->
    <logger class="com.android.tradefed.log.FileLogger">
        <option name="log-level" value="VERBOSE" />
        <option name="log-level-display" value="VERBOSE" />
    </logger>

    <!-- Save logs to file system -->
    <log_saver class="com.android.tradefed.result.FileSystemLogSaver" />

    <!-- Show results on screen and in files -->
    <result_reporter class="com.android.tradefed.result.ConsoleResultReporter" />
    <result_reporter class="com.android.tradefed.result.suite.SuiteResultReporter" />
    <result_reporter class="com.android.tradefed.result.MetricsXMLResultReporter"/>
</configuration>
```

Would you like a **real-world sample** XML that installs an APK and runs it?

Templates and includes in Tradefed XML configuration

bookmark_border
Creating one monolithic XML configuration to define a test is not always practical. If you want to re-use part of the setup to run similar tests, you would be forced to copy and maintain two giant XML files.

This is where template and include tags in Tradefed XML Configuration definition come in handy. They allow you to set placeholders in some XML configuration to add part of another XML configuration.

Example definition for templates

<configuration description="Common base configuration for local runs with minimum overhead">
  <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

  <template-include name="preparers" default="empty" />

  <template-include name="test" default="empty" />

  <template-include name="reporters" default="empty" />
</configuration>
Templates are placeholders with a name to reference them, and an optional default field. The default field defines the default replacement XML that should be used.

In order to replace a template for a given configuration, the following command parameter

needs to be added to the command line:

--template:map <name of template>=<replacement XML config path>

--template:map preparers=empty
For example:

<template-include name="preparers" default="empty" />
The empty reference in this case refers to the empty.xml configuration that contains nothing; we use it as our reference to replace with nothing.

The path of XML configs can be absolute or relative to the res/config folder inside Tradefed's JAR resources. Here are a few of their locations:

tools/tradefederation/core/res/config
tools/tradefederation/core/tests/res/config
tools/tradedeferation/contrib/res/config
Example definition of includes

<configuration description="Common base configuration for local runs with minimum overhead">
  <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

  <include name="empty"/>
</configuration>
Includes are simpler than Templates as they require no command line arguments; they directly expand the referenced XML in the name tag. Similar to templates, the path to the config can be absolute or relative. Still, for includes we recommend using only relative paths as they are more portable in Tradefed. Absolute paths would not be valid if Tradefed is moved to another machine.

Misconfiguration
In case of misconfiguration, such as when the replacement XML cannot be found, Tradefed will throw a ConfigurationException with description of what seems to be missing or misconfigured.

In **simple words**, Tradefed's **template** and **include** features help you **reuse** parts of XML test configurations, so you don't have to repeat the same content in multiple files.

🔧 Why are they useful?

When you write tests, many parts like **`build_provider`**, **`target_preparer`**, **`reporters`**, etc., may be **common across multiple test cases**.

Instead of copying-pasting the same XML code everywhere, you can **write it once** and **reuse it using templates or includes**.

📄 1. `template-include` (Dynamic placeholder)

It defines **named placeholders** in the XML config, and you decide what to insert at **runtime** using a command line flag.

✅ Example: Template config (`base_config.xml`)

```
xml

<configuration description="Basic config with placeholders">
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

    <template-include name="preparers" default="empty" />
    <template-include name="test" default="empty" />
    <template-include name="reporters" default="empty" />
</configuration>
```

This XML has 3 **placeholders**:

- **`preparers`**
- **`test`**
- **`reporters`**

They are initially set to **`"empty"`**.

🔧 To use them, you run Tradefed like this:

```
sh

tradefed.sh run commandAndExit \
  --template:map test=example_test.xml \
  --template:map reporters=console_reporter.xml \
  base_config.xml
```

It means:
👉 Replace **`<template-include name="test"/>`** with contents of **`example_test.xml`**
👉 Replace **`<template-include name="reporters"/>`** with contents of **`console_reporter.xml`**

✅ `empty.xml`

Tradefed comes with an **`empty.xml`** like this:

```xml
<configuration description="Empty placeholder">
</configuration>
```

Useful if you don't want to insert anything.

📄 2. `include` (Static inclusion)

This directly **injects the content** of another XML **at parse time**. No command-line args needed.

✅ Example:

```xml
<configuration description="Local test setup">
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />
    <include name="empty" />
</configuration>
```

This will **expand** whatever is inside **`empty.xml` at the position of `<include>`**.

🧠 Difference Between `template-include` and `include`

Feature	`template-include`	`include`
Replaced at	**Runtime** via **`--template:map`**	**Parse time** (static)
CLI required?	✅ Yes	❌ No
Use case	Reuse & replace based on context	Direct, simple reuse
Default fallback	✅ Can use **`default="empty"`**	❌ No default option

💥 If there's a mistake?

If Tradefed can't find the XML file or template, it throws a:

```lua
ConfigurationException: Could not find configuration named 'xyz'
```

🧪 Summary

Think of this like creating **functions or modules** in programming:

- **`template-include`** = parameterized function (you can change behavior via CLI)
- **`include`** = copy-paste a helper file into main code

Let me know if you want a full working XML example using both!

Tradefed configuration object

bookmark_border
Tradefed XML configuration is parsed, and a Configuration object is created from it that describes the complete configuration.

The object is described by the IConfiguration interface. it will contain an instance of all the objects defined in the XML.

This example:

```
<configuration description="<description of the configuration>">
    <!-- A build provider that takes local device information -->
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

    <!-- Some target preparation, disabled by default -->
    <target_preparer class="com.android.tradefed.targetprep.PreloadedClassesPreparer">
        <option name="disable" value="true" />
    </target_preparer>

    <!-- One test running some unit tests -->
    <test class="com.android.tradefed.testtype.HostTest">
        <option name="class" value="com.android.tradefed.build.BuildInfoTest" />
    </test>
</configuration>
```

Will result in:

IConfiguration#getBuildProvider() to return a BootstrapBuildProvider instance.
IConfiguration#getTargetPreparers() to return a list of ITargetPreparer containing an instance of PreloadedClassesPreparer.
IConfiguration#getTests() to return a list of IRemoteTest containing an instance of HostTest.
Every single object in the configuration object can be mapped to the XML definition, so understanding the XML definition usually helps to understand what to expect from the configuration object.

Let’s break this down into **very simple terms**:

❓ What is a Tradefed Configuration Object?

When you write a **Tradefed XML configuration**, it defines:

- how to build the test,
- how to prepare the device,
- and what tests to run.

💡 **Tradefed reads this XML**, and turns it into a **Java object** in memory — called the **`Configuration`** object.

This object **implements** the interface **`IConfiguration`**, and it holds references to all the components you described in the XML.

✅ Example (Simple version)

Here's an example XML:

```xml
xml

<configuration description="Example config">
    <build_provider class="com.android.tradefed.build.BootstrapBuildProvider" />

    <target_preparer class="com.android.tradefed.targetprep.PreloadedClassesPreparer">
        <option name="disable" value="true" />
    </target_preparer>

    <test class="com.android.tradefed.testtype.HostTest">
        <option name="class" value="com.android.tradefed.build.BuildInfoTest" />
    </test>
</configuration>
```

🔍 What happens when Tradefed loads this?

It builds this object in memory:

- **`getBuildProvider()`** → gives you a **`BootstrapBuildProvider`** object.
- **`getTargetPreparers()`** → gives you a list that contains a **`PreloadedClassesPreparer`** object.
- **`getTests()`** → gives you a list that contains a **`HostTest`** object.

Each part of the XML is **converted into a Java object**, and stored in the configuration.

🧠 Why is this important?

- You can write flexible XML files for test setup.
- Tradefed knows what to build, prepare, and test.

- Developers can **dynamically change configuration** or **debug** it by inspecting the `IConfiguration` object.

🧩 Summary

XML Tag	Turns Into Java Object
`<build_provider>`	`IBuildProvider`
`<target_preparer>`	`ITargetPreparer`
`<test>`	`IRemoteTest`
`<option>`	Sets values inside the above objects

Let me know if you want a visual or code demo too!

> yes

Here's a **simple visual and explanation** of how a **Tradefed XML configuration** turns into a **Java object structure**:

🗒️ Tradefed XML Config Example

```xml

<configuration description="Simple demo config">

    <build_provider class="com.android.tradefed.build.LocalBuildProvider" />

    <target_preparer class="com.android.tradefed.targetprep.FastbootDeviceFlasher">
        <option name="wipe-data" value="true" />
    </target_preparer>

    <test class="com.android.tradefed.testtype.InstrumentationTest">
        <option name="package" value="com.example.myapp" />
    </test>

</configuration>
```

📦 Tradefed Internally Builds This Java Object:

```yaml

Configuration (implements IConfiguration)
├── buildProvider: LocalBuildProvider
```

```
├── targetPreparers: [FastbootDeviceFlasher]
│   └── option: wipe-data = true
└── tests: [InstrumentationTest]
    └── option: package = com.example.myapp
```

🔄 How It Maps:

XML Component	Java Interface	What it does
`<build_provider>`	**`IBuildProvider`**	Defines **how to get the test build**
`<target_preparer>`	**`ITargetPreparer`**	**Prepares device** (flash, root, etc.)
`<test>`	**`IRemoteTest`**	Defines **which test to run**
`<option>`	Sets a field inside above components	e.g., **`wipe-data=true`**

✅ Access in Code

Once Tradefed parses the XML:

```java
IConfiguration config = ... // gets built from XML

IBuildProvider provider = config.getBuildProvider(); // LocalBuildProvider
List<ITargetPreparer> preparers = config.getTargetPreparers(); // FastbootDeviceFlasher
List<IRemoteTest> tests = config.getTests(); // InstrumentationTest
```

💡 Why This Is Useful

- You can **create, reuse, and modify** test setups easily.
- You can mix and match modules (e.g., run **`InstrumentationTest`** with different device flashers).
- Easy to **debug** since config is fully introspectable via code.

Let me know if you'd like:

- A real **`.xml`** config for CTS/VTS
- A minimal Java code sample that loads a config
- Details on how **`<option>`** values map to Java fields

↪

A real `.xml` config for CTS/VTS

yes

Great! Here's a **real example** of a simple **CTS-style Tradefed XML configuration file**, with clear annotations to help you understand what each part does.

✅ Real Tradefed XML Config for CTS/VTS-like Test

📄 **`cts-config.xml`**:

```
xml

<configuration description="CTS Sample Test Config">

    <!-- Build provider: tells TF how to get the build to test -->
    <build_provider class="com.android.tradefed.build.LocalFolderBuildProvider">
        <option name="root-dir" value="/tmp/cts-build" />
    </build_provider>

    <!-- Target preparer: flashes and prepares device for testing -->
    <target_preparer
class="com.android.compatibility.common.tradefed.targetprep.CompatibilityDevicePreparer">
        <option name="force-root" value="true" />
        <option name="wipe-data" value="true" />
    </target_preparer>

    <!-- Test module: run host-side JUnit tests -->
    <test class="com.android.compatibility.common.tradefed.testtype.JarHostTest">
        <option name="jar" value="cts/tests/sample/SampleTests.jar" />
        <option name="class" value="com.example.SampleTest#testSomething" />
    </test>

</configuration>
```

🧠 What this means

XML Section	What It Does
`<build_provider>`	Tells TF to use files from **`/tmp/cts-build`** as the build to test. No need to download.
`<target_preparer>`	Wipes the device and enables root before the test.
`<test>`	Runs a JUnit-style test inside a JAR file on the **host side** (not on the device). Only runs the method **`testSomething()`** from **`SampleTest`** class.

🔍 Behind the scenes: Tradefed creates

```
java

IConfiguration config = ...; // From cts-config.xml

IBuildProvider build = config.getBuildProvider();  // LocalFolderBuildProvider
```

```
List<ITargetPreparer> prep = config.getTargetPreparers(); // CompatibilityDevicePreparer
List<IRemoteTest> test = config.getTests(); // JarHostTest
```

🪛 What types of CTS/VTS tests use these

Test Type	Test Class Used	Runs on
Device-side instrumentation	**`com.android.tradefed.testtype.InstrumentationTest`**	On-device
Host-side JUnit (e.g., CTS)	**`JarHostTest`**	On-host
GTest (native C++)	**`NativeTest`**	On-device
Compatibility tests	**`CompatibilityTest`**	Both host and device
Kernel tests (VTS)	**`VtsMultiDeviceTest`**	On-device (multi-device aware)

Would you like me to show:

- A **device-side test** example (Instrumentation)?
- A **VTS multi-device** XML config?
- A **Java code snippet** to run this config from code?

Let me know!

> but these config already provided for cts and vts, then why we need these?

Excellent question! You're absolutely right — **CTS** and **VTS** already provide pre-written Tradefed configuration XML files. So, why do we ever need to write our own?

Let me explain in **simple words with clear use-cases**:

✅ Why we might need our own Tradefed config

Reason	Explanation
1. Create Custom Tests	If you're adding your own module, app, or HAL and want to test it, you'll need your own XML to describe how to run it.
2. Local Testing	You may want to run **just a small part** of CTS/VTS locally — instead of full test plans, you can create a tiny config for one test.
3. Internal/Non-CTS Tests	For **device bring-up**, platform development, or vendor HAL testing, you may create tests **not intended for CTS/VTS**.

Reason	Explanation
4. Debugging	Want to run a test with **root access**, **logs**, or a **different environment**? You can make your own XML with special options.
5. Combine multiple steps	Want to run a test, then reboot, then run another? You can chain multiple steps in your config. CTS configs don't support this directly.

🔧 Example: When you might write your own config

Let's say:

- You're adding a new camera HAL.
- You wrote a GTest: `/data/local/tmp/test_camera_hal`
- You want to flash the build, push the test, run it, and collect logs.

➡️ You'd write a `my_camera_test.xml` like:

```
xml

<configuration description="Camera HAL GTest">

    <build_provider class="com.android.tradefed.build.LocalFolderBuildProvider">
        <option name="root-dir" value="/tmp/my_build" />
    </build_provider>

    <target_preparer class="com.android.tradefed.targetprep.PushFilePreparer">
        <option name="push" value="test_camera_hal->/data/local/tmp/test_camera_hal" />
    </target_preparer>

    <test class="com.android.tradefed.testtype.GTest">
        <option name="native-test-device-path" value="/data/local/tmp/test_camera_hal" />
    </test>

</configuration>
```

Then run:

```
bash

tradefed.sh run commandAndExit cts-configs/my_camera_test.xml
```

🤔 Summary

CTS/VTS Config	Your Own Config
Predefined and rigid	Fully customizable
Runs official Android tests	Can run any kind of test
Usually full test plans	Can be minimal (1 test)
Not editable (without rebuilding)	Just write XML and run

Would you like a **step-by-step guide** on how to create and run your own test config locally using Tradefed?

Or maybe how to convert a **JUnit or instrumentation test into a working config**?

Let me know how you'd like to go deeper.

> i'm talking about these config already provided for cts and vts, then why we need these again for cts and vts?

Great clarification. You're asking:

> **CTS and VTS already come with their own Tradefed XML configurations — so why would we ever need to write *more* XML configs when working with CTS or VTS themselves?**

Let me explain **in simple terms**:

🔁 Short Answer:

Even though **CTS/VTS provide default Tradefed configs**, sometimes we still need to create or modify configs to:

1. **Run only specific tests (subset)**
2. **Run with custom device setup or flags**
3. **Debug issues or rerun failed tests**
4. **Extend tests for custom hardware/modules**
5. **Run tests in CI/automated pipelines differently**

📦 What CTS/VTS provides by default

- They come with **predefined XML config files**, like:

```
bash


android-cts/testcases/android.hardware.camera.cts.xml
vts/testcases/vts_hal_camera_target_test.xml
```

- These describe:
  - What test to run
  - What build to use

- What target_preparer/setup is needed

These are good for **default, full test runs**.

🚧 But sometimes you still need custom XML even in CTS/VTS

Here's **why you might need additional configs** even for CTS/VTS:

Scenario	Why extra config is needed
🔬 **Debug one specific test**	You want to isolate and debug only one test class or method, not run the full suite.
⚙️ **Custom device behavior**	Need to **skip reboot**, **preserve logcat**, or add special **`target_preparer`** steps.
🔪 **Subset test runs**	Run only a specific module like **`android.hardware.camera.cts.CameraTestCases`**.
🔁 **Retry failed tests**	You create a retry config with just the failed tests.
🧩 **New HAL or custom modules**	CTS/VTS has generic tests, but you may need to add tests for **your vendor extension** (e.g. custom HAL service).
🛰️ **CI/CD Automation**	You may write a config that integrates with your lab setup, pushing logs to server, etc.

🛠️ Example

CTS config:

```xml
<test class="com.android.tradefed.testtype.HostTest">
  <option name="class" value="android.hardware.camera.cts.CameraTestCases" />
</test>
```

You want to run only **`testVideoRecording`** in debug mode → you copy this config and add:

```xml
<option name="method" value="testVideoRecording" />
```

Now you can debug faster and run this single method without launching the entire CTS suite.

✅ Summary

Even though **CTS/VTS provide XML configs**, you still need to create your own **in these cases**:

- To **debug faster**
- To **run a subset of tests**

- To **change device setup**
- To **extend tests** for custom hardware
- To **automate** differently in labs

Let me know if you want a working example of how to:

- Extract and modify an existing CTS config
- Run only selected classes/methods
- Inject feature flags or options

I'll guide you step by step.

> where to find the config files of cts and vts

Good question! Here's where to find the **XML configuration files** for **CTS** and **VTS** inside the AOSP (Android source tree):

📍 CTS Configuration Files Location

CTS configs are found in:

```
bash

<android-root>/cts/
```

Common XML paths:

File	Purpose
`cts/testcases/*.xml`	Main test module configs
`cts/hostsidetests/*/AndroidTest.xml`	Host-side test configs
`cts/tests/*/AndroidTest.xml`	Device-side test configs
`cts/Android.mk` / **`Android.bp`**	Build definitions for modules

Example:

- Camera CTS config:

```bash
cts/testcases/android.hardware.camera.cts.xml
```

- Host-side test for camera:

```bash
cts/hostsidetests/camera/AndroidTest.xml
```

📍 VTS Configuration Files Location

VTS configs are found in:

```bash
<android-root>/test/vts/
```

Common XML paths:

File	Purpose
`test/vts/testcases/*/*.xml`	VTS test module configs
`test/vts/tools/tradefed/*`	VTS test runner tools/scripts

Example:

- Camera HAL test:

```swift
test/vts/testcases/hal/camera/functional/VtsHalCameraV2_1TargetTest.xml
```

- VTS shell test:

```bash
test/vts/testcases/kernel/syscalls/VtsKernelSyscallTest.xml
```

🧭 How to Search for XML Configs Quickly

Use **`find`** or **`grep`** from the root of your Android source:

```bash
# List all CTS test configs
find cts/ -name "*.xml"

# List all VTS test configs
find test/vts/ -name "*.xml"

# Or search inside for a class name
grep -r "HostTest" cts/
```

🔍 Optional: Tradefed Binary Location

When you want to **run CTS/VTS configs** using Tradefed manually:

- CTS command:

  ```bash
  cts-tradefed
  ```

- VTS command:

  ```bash
  vts-tradefed
  ```

They will use these XML files to configure the tests to run.

✅ Summary

Framework	Config Location
CTS	**`cts/testcases/*.xml`**, **`cts/hostsidetests/*/AndroidTest.xml`**
VTS	**`test/vts/testcases/*/*.xml`**

You can customize or clone these XML files to:

- Run selected tests
- Add target preparers
- Inject options

Let me know if you'd like to extract one and run it manually via **`cts-tradefed`** or **`vts-tradefed`**. I can show an example.

cts-tradefed and vts-tradefed

Great! Let's explain `cts-tradefed` and `vts-tradefed` in **simple words** with **examples**:

✅ What are `cts-tradefed` and `vts-tradefed`?

They are **test runners** (command-line tools) built on **Tradefed** (Test Framework) that run **CTS** (Compatibility Test Suite) or **VTS** (Vendor Test Suite).

They:

- Parse XML config files (modules)
- Set up the device/environment
- Run test cases
- Report results

🔪 `cts-tradefed` (for Compatibility Tests)

- **Command** to start CTS interactive shell:

```
bash

cts-tradefed
```

- This opens the CTS console:

```
arduino

CTS console: run cts >
```

- Example to run a CTS module:

```
bash

run cts --module CtsCameraTestCases
```

- Example to run a specific test class:

```bash
bash

run cts --class android.hardware.camera2.cts.CameraTest
```

🧪 `vts-tradefed` (for Vendor Tests)

- **Command** to start VTS interactive shell:

```bash
bash

vts-tradefed
```

- This opens the VTS console:

```arduino
arduino

VTS console: run vts >
```

- Example to run a VTS HAL test:

```bash
bash

run vts --module VtsHalCameraV2_1TargetTest
```

- Example to run a kernel syscall test:

```bash
bash

run vts --module VtsKernelSyscallTest
```

🔁 Behind the scenes

When you type **`run cts --module ...`**, it:

1. Loads the XML config file
2. Instantiates **`BuildProvider`**, **`TargetPreparer`**, and **`Test`** objects
3. Pushes files or prepares environment
4. Runs tests using instrumentation or host-side test runners
5. Collects results (as logs or reports)

🗂️ Where are they in AOSP?

- CTS tool:

```
bash

prebuilts/cmdline-tools/cts-tradefed
```

- VTS tool:

```
bash

test/vts/tools/vts-tradefed
```

They are built automatically when you run:

```
bash

make cts
make vts
```

🧠 Summary

Tool	Runs	Entry	Example
`cts-tradefed`	Compatibility tests (AOSP behavior)	**`run cts`**	**`run cts --module CtsAppSecurityTestCases`**
`vts-tradefed`	Vendor HAL/kernel tests	**`run vts`**	**`run vts --module VtsHalCameraV2_1TargetTest`**

Would you like a sample test run and what the output looks like?

> why we need these although we already have tradefed

Great question! Here's a **simple and clear** explanation:

🤔 If we already have Tradefed, why do we need `cts-tradefed` and `vts-tradefed`?

Because **`cts-tradefed`** and **`vts-tradefed`** are **specialized wrappers** around Tradefed — tailored to run **specific sets of tests (CTS/VTS)** with preloaded configurations, setup, and environment.

🧱 Think of it like this:

- **Tradefed** = A general-purpose test engine (like a raw car engine)
- **cts-tradefed** = A car built from that engine for road compatibility testing (CTS)
- **vts-tradefed** = A different car from that same engine for vendor-level testing (VTS)

They **use Tradefed under the hood**, but:

- Load **different config files**
- Use **different test modules**
- May set up devices differently
- Have **different command options**

✅ Why not just use Tradefed directly?

You **can**, but you'd have to:

Task	Using Raw Tradefed	Using `cts-tradefed` / `vts-tradefed`
Load test modules	Manually specify XMLs	Pre-configured
Set up device prep steps	Manually add preparers	Already built-in
Build filters and options	Manual and error-prone	Pre-bundled logic
Understand full internals	Required	Not required

In short:

- `cts-tradefed` and `vts-tradefed` = ready-made, plug-and-play tools for CTS/VTS
- They **simplify**, **automate**, and **standardize** testing

---

🔪 Example difference

```
bash


# Using Tradefed directly
tradefed.sh run config cts.xml

# Using CTS wrapper (easier!)
cts-tradefed
run cts --module CtsAppSecurityTestCases
```

🧠 Summary

Tool	Purpose	Based on	What's different?
Tradefed	Generic test framework	Core engine	Requires manual setup
cts-tradefed	Compatibility tests for Android devices	Tradefed	Preconfigured for CTS
vts-tradefed	Vendor HAL/kernel tests	Tradefed	Preconfigured for VTS

Let me know if you'd like to see how a CTS/VTS config XML maps into test objects!

how to run cts and vts

Here's a **simple and complete guide** on how to **run CTS and VTS** using `cts-tradefed` and `vts-tradefed`.

✅ Preconditions (for both CTS and VTS):

1. 🔌 **Device connected** via USB or TCP (ADB must detect it):

```
bash

adb devices
```

2. ✅ **Device is user-debug/user build**, matching CTS/VTS requirements.
3. 🛠️ **CTS/VTS test suite downloaded** (from AOSP build or official source).
4. ⚙️ Optional: Ensure `JAVA_HOME`, `ANDROID_BUILD_TOP` are set if built from source.

🧪 1. Run CTS

💡 What is CTS?

Compatibility Test Suite — tests **framework APIs**, **behavior**, and **compatibility**.

▶️ Steps:

1. **Start CTS test harness**

```
bash

./android-cts/tools/cts-tradefed
```

2. **Basic full run**

```
bash

run cts
```

3. **Run specific module**

```
bash

run cts --module CtsAppSecurityHostTestCases
```

4. **Run specific test class or method**

```
bash

run cts --module CtsAppSecurityHostTestCases --class com.android.cts.security.HostTest
```

5. **Run on specific device**

```
bash

run cts --serial <DEVICE_SERIAL>
```

6. **Check available modules**

```
bash

list modules
```

⚙️ 2. Run VTS

💡 What is VTS?

Vendor Test Suite — tests **HALs**, **kernel**, **binder services**, **VNDK**, etc.

▶️ Steps:

1. **Start VTS test harness**

```
bash

./android-vts/tools/vts-tradefed
```

2. **Basic full run**

```
bash

run vts
```

3. **Run specific module**

```
bash

run vts --module VtsHalCameraProviderV2_4TargetTest
```

4. **List available modules**

```
bash

list modules
```

5. **Run only kernel tests**

```
bash

run vts --plan VtsKernelTestPlan
```

📦 Output and Reports

- Results are saved under:

```
bash

./android-cts/results/ or ./android-vts/results/
```

- Reports are HTML/XML format with **pass/fail logs**.

🛑 Troubleshooting

Issue	Solution
Device not detected	**`adb devices`**, reboot, or check USB
`Unknown module`	Use **`list modules`** to check spelling
Tests skipped	Device may be in **`user`** mode or lacks root/debug
Tradefed crashes	Check Java version or logs in **`stderr.log`**

✅ Summary

Test Suite	Command
Start CTS	`./android-cts/tools/cts-tradefed`
Run all CTS tests	`run cts`
Start VTS	`./android-vts/tools/vts-tradefed`
Run all VTS tests	`run vts`

Let me know if you'd like help filtering by ABI, test class, or running parallel tests!

Mobly test framework

bookmark_border
Mobly is an open-source test framework for host-driven, end-to-end automated testing. You can use Mobly to orchestrate actions across multiple devices to validate Android features including Wi-Fi and telephony.

Mobly basics
Using Mobly, you can write and execute Mobly tests to verify Android features. To learn the basics of using Mobly, see the Mobly tutorial.

You can also use the Mobly Snippet Lib to build custom test APKs for triggering device-side code from the host.

Mobly features
The Mobly test framework provides the following features:

Flexible test organization

Mobly is a standalone library, on which users can build different test suites. Mobly users are isolated from other users' changes.

Users can add dependencies to project-specific test suites without impacting other users.

Mobly snippets lets users create custom test APKs with only the APIs relevant to the test. This minimizes the file size overhead of test APKs.

Simple to maintain and debug

Mobly provides informative test results including test and snippet stacktraces, and per-test-iteration results for repeated or retried tests.

The Mobly codebase is covered by presubmit unit tests, and enforces unit test coverage for new code contributions, which reduces the risk of code errors.

Mobly snippet APKs can run without privileged access, which allows testing on a large number of platforms.

Example Mobly tests
The following are examples of Mobly tests for Android features:

Test for Wi-Fi Direct
Test for CompanionDeviceManager

Here's a **clear and simple explanation** of the **Mobly Test Framework**, its purpose, and how to use it.

✅ What is Mobly?

Mobly is an **open-source Python-based test framework** developed by Google.
It is used for **automated, end-to-end testing** of Android features, especially those that involve:

- 📶 **Wi-Fi**
- 📱 **Telephony**
- 📡 **Bluetooth**
- 🔌 **Multi-device interaction (host-driven)**

🔧 Key Features of Mobly

Feature	Description
🔀 **Host-driven**	Tests are written and run on a **host machine** (your PC), not directly on the Android device.
📦 **Snippet support**	You can build **custom APKs** (Mobly snippets) that run device-side code triggered from the host.
🗂️ **Modular & Isolated**	Each test suite can have its own dependencies without interfering with others.
🐞 **Easy Debugging**	Gives **detailed per-test logs**, **stack traces**, and **iteration-level** info.
🔓 **No root needed**	Most tests and snippets can run without root access.

🔪 Mobly Test Structure

Mobly tests are just **Python scripts** that follow a certain structure.

Example Test:

```python
from mobly import base_test
from mobly import test_runner

class MySimpleTest(base_test.BaseTestClass):
    def setup_class(self):
        self.my_device = self.android_devices[0]
        self.my_device.adb.shell('input keyevent 26')  # Turn screen on

    def test_reboot_device(self):
        self.my_device.reboot()
        assert self.my_device.adb.shell('getprop sys.boot_completed') == '1'

if __name__ == '__main__':
    test_runner.main()
```

🧧 Mobly Snippet Lib

Sometimes, you need to **call custom Java code** on the Android device from your Mobly Python test.

You can:

1. Write a custom Android service (a test APK)
2. Use **Mobly Snippet Lib** to:
    - Expose Java methods
    - Trigger them from Python

🔁 Think of it like **RPC between Python and Java APK**.

📦 Installation

Install Mobly via pip:

```bash
pip install mobly
```

▶️ Running a Mobly Test

1. Create a **`test_config.yaml`** file:

```yaml
TestBeds:
  - Name: MyTest
    Controllers:
```

```
      AndroidDevice:
        - serial: <device_serial>
```

2. Run the test:

```
bash

python my_mobly_test.py -c test_config.yaml
```

📑 Where It's Used

Mobly is used in Android for:

Area	Example
Wi-Fi	Wi-Fi Direct, AP scanning
Bluetooth	Pairing, audio
Telephony	Calling, SMS
Multi-device	CompanionDeviceManager

✅ Summary

Component	Role
Host Python Test	Main logic
Mobly Snippet APK	Optional device-side service
ADB	Controls device
YAML config	Defines test environment

Let me know if you want a **working Mobly test example** or want to build a **custom Mobly snippet APK**!

> confusing

No worries — let me break down **Mobly** in the **simplest way possible**, step by step:

🟢 What is Mobly?

Mobly is a **Python tool** used to **test Android devices from your PC**.

Think of it like:

> "A robot script written in Python that controls your phone using ADB and checks if things are working."

🧠 Why use Mobly?

Because you want to **test features like Wi-Fi, calls, Bluetooth**, etc., automatically.

📱 Where does the code run?

- **Python code runs on your PC**
- It talks to your Android phone using:
  - **ADB** (command line)
  - Optionally a **test APK** called a "snippet" (if you need Java code)

🧪 How does a Mobly test look?

A very basic test might look like:

```
python

from mobly import base_test, test_runner

class HelloMobly(base_test.BaseTestClass):
    def test_check_boot_completed(self):
        output = self.android_devices[0].adb.shell("getprop sys.boot_completed")
        assert output.strip() == "1"

if __name__ == '__main__':
    test_runner.main()
```

This checks if the phone has finished booting. It uses **`adb shell`** in Python.

📄 What else do you need?

You also need a **config file** (YAML) to tell Mobly which phone to test:

```
yaml

TestBeds:
  - Name: MyPhone
    Controllers:
      AndroidDevice:
        - serial: <your-device-serial>
```

Save it as **`test_config.yaml`**.

▶️ How to run it:

```
bash

python hello_mobly.py -c test_config.yaml
```

That's it.

💡 When to use Mobly Snippet APK?

If Python + ADB is not enough and you need Java/Android code to:

- Access internal APIs
- Run something inside the phone

Then you build a special test APK called a **Mobly Snippet APK**, install it, and trigger it from Python.

🧵 Summary

Item	Description
Mobly	A Python tool to control Android phones
Runs on	Your computer
Controls	Android via ADB
Optional	Mobly snippet APK (Java code on device)
Best for	Wi-Fi, Bluetooth, Telephony, multi-device testing

Would you like a **step-by-step working example** that you can run now?

> yes

Great! Here's a **step-by-step working Mobly test** example that you can run on your Android phone using just Python and ADB — **no APKs required**.

✅ What You'll Do

You'll write a Mobly test that:

- Connects to your Android phone
- Checks if the phone has finished booting
- Prints a pass/fail result

🧰 Prerequisites

1. **Python 3.6+**
2. **ADB installed** and your phone connected (**`adb devices`**)
3. Install Mobly:

```
bash

pip install mobly
```

🗂️ Step 1: Create Test Script

Create a file called **`hello_mobly.py`**:

```
python

from mobly import base_test, test_runner

class HelloMoblyTest(base_test.BaseTestClass):
    def test_check_boot_completed(self):
        output = self.android_devices[0].adb.shell("getprop sys.boot_completed")
        assert output.strip() == "1", "Phone has not finished booting"

if __name__ == '__main__':
    test_runner.main()
```

📄 Step 2: Create Config File

Create a file called **`test_config.yaml`**:

```
yaml

TestBeds:
  - Name: MyAndroidDevice
    Controllers:
      AndroidDevice:
        - serial: <YOUR_DEVICE_SERIAL>
```

👉 Run **`adb devices`** to get your device serial number, and replace **`<YOUR_DEVICE_SERIAL>`** with that.

Example:

```
yaml

        - serial: RZ8N81XXXXY
```

▶️ Step 3: Run the Test

In your terminal:

```
bash

python hello_mobly.py -c test_config.yaml
```

You should see output like:

```
yaml

2025-07-22 22:15:45 INFO mobly.base_test: HelloMoblyTest: test_check_boot_completed PASS
```

🧪 Result

If the phone has booted, it **passes**. If not, it **fails** with:

```
makefile

AssertionError: Phone has not finished booting
```

🧠 What You Learned

- Mobly uses Python + ADB
- Test connects to phone and runs a command
- No need for APK if you're testing using ADB shell commands

Would you like to try more tests — like **Wi-Fi ON/OFF**, **airplane mode**, or one using a **snippet APK**?

> yes

Awesome! Let's expand step-by-step with **Mobly tests for common Android features** using just ADB (no APK yet). We'll write **3 Mobly tests**:

✅ New Test: Wi-Fi Toggle Test

This test will:

1. Turn **Wi-Fi OFF**
2. Wait 2 seconds
3. Turn **Wi-Fi ON**
4. Check Wi-Fi is ON

🗂️ Step 1: Update Your Mobly Test Script

Name it **`wifi_toggle_test.py`**:

```
python

import time
from mobly import base_test, test_runner

class WifiToggleTest(base_test.BaseTestClass):
    def test_wifi_toggle(self):
        ad = self.android_devices[0]

        print("Turning Wi-Fi OFF...")
        ad.adb.shell("svc wifi disable")
        time.sleep(2)

        print("Turning Wi-Fi ON...")
        ad.adb.shell("svc wifi enable")
        time.sleep(2)

        # Check Wi-Fi is ON
        wifi_status = ad.adb.shell("dumpsys wifi | grep 'Wi-Fi is'").strip()
        print("Wi-Fi status:", wifi_status)
        assert "enabled" in wifi_status.lower(), "Wi-Fi is not enabled"

if __name__ == '__main__':
    test_runner.main()
```

📄 Step 2: Reuse `test_config.yaml`

Same file as before:

```
yaml

TestBeds:
  - Name: MyAndroidDevice
    Controllers:
```

```
    AndroidDevice:
      - serial: <YOUR_DEVICE_SERIAL>
```

▶️ Step 3: Run the Test

```
bash

python wifi_toggle_test.py -c test_config.yaml
```

✅ If Wi-Fi is ON at the end, test passes.
❌ If not, you'll get:

```
vbnet

AssertionError: Wi-Fi is not enabled
```

🔁 Want More?

Here are two more test ideas you can run **without any snippet APK**:

📶 Airplane Mode Test (ADB shell setting)

```
python

    def test_airplane_mode_toggle(self):
        ad = self.android_devices[0]

        # Enable airplane mode
        ad.adb.shell("settings put global airplane_mode_on 1")
        ad.adb.shell("am broadcast -a android.intent.action.AIRPLANE_MODE --ez state true")
        time.sleep(2)

        # Disable airplane mode
        ad.adb.shell("settings put global airplane_mode_on 0")
        ad.adb.shell("am broadcast -a android.intent.action.AIRPLANE_MODE --ez state false")
        time.sleep(2)

        result = ad.adb.shell("settings get global airplane_mode_on").strip()
        assert result == "0", "Airplane mode is still enabled"
```

📱 Check Battery Percentage

```
python

    def test_battery_level(self):
        ad = self.android_devices[0]
        level = ad.adb.shell("dumpsys battery | grep level").strip()
```

```
        print("Battery level:", level)
        assert "level" in level
```

🧠 Summary

Test Name	What it Does
`test_wifi_toggle`	Turns Wi-Fi OFF and ON
`test_airplane_mode`	Toggles airplane mode
`test_battery_level`	Reads battery percentage

Would you like to move to **Mobly Snippet APK** (for custom Java-side code), or want **more shell-based Mobly tests**?

↳
Would you like to move to **Mobly Snippet APK** (for custom Java-side code),
this one

Great! Let's now explore **Mobly Snippet APKs** — these let you write **custom Java code that runs on the Android device**, and control it from your **Mobly test (Python on host)**.

✅ What is a Mobly Snippet?

A **Mobly Snippet** is:

- A **small Android app** (APK) with custom test APIs you define in Java.
- It acts like a **bridge** between Mobly on your PC (Python) and the phone (Java).
- You call Java methods from Python using `.rpc`.

🧱 Overview of Steps

We'll create:

1. A minimal **Java-based Mobly Snippet APK**
2. A Mobly **Python test** to control it
3. A **config** file to link devices

🔪 Example Use Case: Get Device Info from Java

This Mobly test will:

- Call a Java method on the phone
- Return Android device info (model, brand, API level)

🔧 Step 1: Java Snippet APK

Java File: `MySnippet.java`

```
java

package com.example.mysnippet;

import com.google.android.mobly.snippet.Snippet;
import com.google.android.mobly.snippet.rpc.Rpc;

@Snippet
public class MySnippet {

    @Rpc
    public String getDeviceInfo() {
        String model = android.os.Build.MODEL;
        String brand = android.os.Build.BRAND;
        int apiLevel = android.os.Build.VERSION.SDK_INT;
        return "Model: " + model + ", Brand: " + brand + ", API: " + apiLevel;
    }
}
```

AndroidManifest.xml:

```
xml

<manifest package="com.example.mysnippet"
          xmlns:android="http://schemas.android.com/apk/res/android">
    <application android:label="MySnippet">
        <service android:name="com.google.android.mobly.snippet.SnippetService"
                 android:exported="true">
            <intent-filter>
                <action android:name="com.google.android.mobly.snippet.action.BIND" />
            </intent-filter>
        </service>
    </application>
</manifest>
```

📦 Build APK (Android Studio or Gradle)

Add Mobly Snippet dependency in **`build.gradle`**:

```
gradle

dependencies {
    implementation 'com.google.android.mobly:mobly-snippet-lib:1.0.3'
}
```

Build and install APK to device:

```
bash

adb install -r app/build/outputs/apk/debug/app-debug.apk
```

🐍 Step 2: Python Mobly Test

File: `device_info_test.py`

```
python

from mobly import base_test, test_runner
from mobly.controllers import android_device

class DeviceInfoTest(base_test.BaseTestClass):
    def setup_class(self):
        self.ads = self.register_controller(android_device)
        self.ad = self.ads[0]
        self.ad.load_snippet(name='snippet', package='com.example.mysnippet')

    def teardown_class(self):
        self.ad.snippet.stop_app()

    def test_get_device_info(self):
        info = self.ad.snippet.getDeviceInfo()
        print("Device Info from snippet:", info)
        assert "Model" in info

if __name__ == '__main__':
    test_runner.main()
```

📄 Step 3: `test_config.yaml`

```
yaml

TestBeds:
  - Name: MoblySnippetTest
    Controllers:
      AndroidDevice:
        - serial: <your_device_serial>
```

▶️ Step 4: Run the Test

```
bash

python device_info_test.py -c test_config.yaml
```

✅ Output (example)

```
text

Device Info from snippet: Model: Pixel 6, Brand: google, API: 34
```

💡 Recap

Part	Purpose
Java snippet	Exposes device-side API (**`@Rpc`**)
Mobly Python test	Calls **`.snippet.getDeviceInfo()`**
Snippet Service	Android-side binder bridge to Mobly

Want help generating the **Android Studio snippet project from scratch**, or shall we write a more complex Java snippet like **Wi-Fi manager or telephony**?

↪
hall we write a more complex Java snippet like **Wi-Fi manager or telephony**?
yes

Great! Let's now create a **more complex Mobly Snippet** example — this time using **Wi-Fi Manager** to control Wi-Fi from Java and access network info via Mobly test.

📲 Goal: Control and Query Wi-Fi Using Mobly Snippet

We'll write a **Mobly Snippet APK** that lets your Python test:

- Turn Wi-Fi ON/OFF
- Check if Wi-Fi is enabled
- Get connected SSID

Then, we'll write the Mobly test that uses it.

🧱 Java Snippet Code (Wi-Fi Control)

File: `WifiSnippet.java`

```
java


package com.example.wifisnippet;

import android.content.Context;
import android.net.wifi.WifiInfo;
import android.net.wifi.WifiManager;

import com.google.android.mobly.snippet.Snippet;
import com.google.android.mobly.snippet.rpc.Rpc;

@Snippet
public class WifiSnippet {
    private final WifiManager wifiManager;

    public WifiSnippet() {
        wifiManager = (WifiManager)
Snippet.getApplication().getSystemService(Context.WIFI_SERVICE);
    }

    @Rpc
    public void setWifiEnabled(boolean enable) {
        wifiManager.setWifiEnabled(enable);
    }

    @Rpc
    public boolean isWifiEnabled() {
        return wifiManager.isWifiEnabled();
    }

    @Rpc
    public String getConnectedSSID() {
        WifiInfo wifiInfo = wifiManager.getConnectionInfo();
        return wifiInfo != null ? wifiInfo.getSSID() : "null";
    }
}
```

🗒️ AndroidManifest.xml

```
xml


<manifest package="com.example.wifisnippet"
          xmlns:android="http://schemas.android.com/apk/res/android"
          android:permission="android.permission.CHANGE_WIFI_STATE">

    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
    <uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

    <application android:label="WifiSnippet">
        <service android:name="com.google.android.mobly.snippet.SnippetService"
                 android:exported="true">
            <intent-filter>
                <action android:name="com.google.android.mobly.snippet.action.BIND"/>
```

```
            </intent-filter>
        </service>
    </application>
</manifest>
```

⚒️ Build Instructions

Use Android Studio, or from terminal:

```
bash


./gradlew assembleDebug
adb install -r app/build/outputs/apk/debug/app-debug.apk
```

🐍 Mobly Python Test Code

File: `wifi_test.py`

```
python


from mobly import base_test, test_runner
from mobly.controllers import android_device
import time

class WifiControlTest(base_test.BaseTestClass):
    def setup_class(self):
        self.ads = self.register_controller(android_device)
        self.ad = self.ads[0]
        self.ad.load_snippet(name='wifi', package='com.example.wifisnippet')

    def teardown_class(self):
        self.ad.wifi.stop_app()

    def test_toggle_wifi_and_check_status(self):
        self.ad.wifi.setWifiEnabled(True)
        time.sleep(3)
        assert self.ad.wifi.isWifiEnabled() is True

        self.ad.wifi.setWifiEnabled(False)
        time.sleep(3)
        assert self.ad.wifi.isWifiEnabled() is False

    def test_get_connected_ssid(self):
        ssid = self.ad.wifi.getConnectedSSID()
        print("Connected SSID:", ssid)
```

📄 Config File: `wifi_config.yaml`

```yaml
TestBeds:
  - Name: WifiTestBed
    Controllers:
      AndroidDevice:
        - serial: <your_device_serial>
```

Replace **`<your_device_serial>`** with **`adb devices`** output.

▶️ Run Mobly Test

```bash
python3 wifi_test.py -c wifi_config.yaml
```

✅ Output Example

```text
Connected SSID: "MyHomeWiFi"
```

🧠 Notes

- For Android 10+ you **must grant location permissions** manually for SSID access.

```bash
adb shell pm grant com.example.wifisnippet android.permission.ACCESS_FINE_LOCATION
```

Would you like me to generate the **full Android Studio project structure** for this Mobly snippet?

↪

from mobly import base_test, test_runner from mobly.controllers import android_device import time class WifiControlTest(base_test.BaseTestClass): def setup_class(self): self.ads = self.register_controller(android_device) self.ad = self.ads\[0\] self.ad.load_snippet(name='wifi', package='com.example.wifisnippet') def teardown_class(self): self.ad.wifi.stop_app() def test_toggle_wifi_and_check_status(self): self.ad.wifi.setWifiEnabled(True) time.sleep(3) assert self.ad.wifi.isWifiEnabled() is True self.ad.wifi.setWifiEnabled(False) time.sleep(3) assert

self.ad.wifi.isWifiEnabled() is False def test_get_connected_ssid(self): ssid = self.ad.wifi.getConnectedSSID() print("Connected SSID:", ssid)
what will be flow here

Here's the **step-by-step flow** of the Mobly test using the custom **`WifiSnippet`**:

🔄 Full Execution Flow Breakdown

1. Mobly starts the test class

When you run:

```
bash

python3 wifi_test.py -c wifi_config.yaml
```

Mobly will:

- Read the YAML config file
- Instantiate the test class **`WifiControlTest`**
- Call its **`setup_class()`** method

---

2. Setup Phase

In `setup_class()`:

```
python

self.ads = self.register_controller(android_device)
```

- Registers Android device(s) via Mobly's controller manager.
- **`self.ads`** becomes a list of connected Android devices.

```
python

self.ad = self.ads[0]
```

- Picks the first device.

```
python

self.ad.load_snippet(name='wifi', package='com.example.wifisnippet')
```

- Launches the installed **Mobly Snippet APK** (`com.example.wifisnippet`) on the device.
- Starts the **snippet service**.
- Connects a **gRPC interface** from Python to Java.
- `self.ad.wifi` now becomes a **proxy object** for calling Java RPC methods.

---

3. Test: `test_toggle_wifi_and_check_status()`

```
python

self.ad.wifi.setWifiEnabled(True)
```

- Calls the Java method `@Rpc setWifiEnabled(true)` inside the snippet.
- Java enables Wi-Fi using `wifiManager.setWifiEnabled(true)`.

```
python

assert self.ad.wifi.isWifiEnabled() is True
```

- Calls `@Rpc isWifiEnabled()` and asserts Wi-Fi is ON.

Then same steps repeat to:

- Turn Wi-Fi OFF
- Verify it's OFF

---

4. Test: `test_get_connected_ssid()`

```
python

ssid = self.ad.wifi.getConnectedSSID()
```

- Calls the Java snippet method `@Rpc getConnectedSSID()`
- Java returns current SSID using `wifiManager.getConnectionInfo().getSSID()`
- Result is printed in Python

---

5. Teardown Phase

```
python

def teardown_class(self):
```

```
    self.ad.wifi.stop_app()
```

- Tells Mobly to **unbind** the snippet and **stop the Java service** cleanly.

🧠 Summary of the Flow

Step	Layer	Action
1	Python	Register and connect Android device
2	Python	Load **`wifi`** snippet from APK
3	Java (Snippet)	Start **`SnippetService`** and expose RPCs
4	Python	Call Java methods via **`self.ad.wifi`**
5	Java	Control Wi-Fi via **`WifiManager`**
6	Python	Run assertions and print results
7	Python	Teardown: Stop snippet service

Would you like me to show this interaction with **logging outputs** (from both Java and Python) for better visualization?

> wifi_test.py
> how is the fow of function and who call whom

Great — let's break down the **function flow of `wifi_test.py`** in Mobly and **who calls whom**, step by step.

📄 File: `wifi_test.py`

Here's the file again, simplified for clarity:

```python
from mobly import base_test
from mobly.controllers import android_device

class WifiControlTest(base_test.BaseTestClass):
    def setup_class(self):
        self.ads = self.register_controller(android_device)
        self.ad = self.ads[0]
        self.ad.load_snippet(name='wifi', package='com.example.wifisnippet')

    def teardown_class(self):
        self.ad.wifi.stop_app()

    def test_toggle_wifi_and_check_status(self):
```

```
        self.ad.wifi.setWifiEnabled(True)
        assert self.ad.wifi.isWifiEnabled() is True
        self.ad.wifi.setWifiEnabled(False)
        assert self.ad.wifi.isWifiEnabled() is False

    def test_get_connected_ssid(self):
        ssid = self.ad.wifi.getConnectedSSID()
        print(f"Connected SSID: {ssid}")
```

🔁 FULL FUNCTION FLOW AND CALL CHAIN

🔹 Mobly Entry Point:

You run the test like this:

```
bash

python3 wifi_test.py -c wifi_config.yaml
```

Mobly internally does:

```
python

BaseTestClass.run()
```

✅ 1. `run()` → Calls → `setup_class()`

- **`Mobly`** framework looks for **`setup_class()`** (defined in your test).
- Calls it to prepare everything.

`setup_class()` does:

```
python

self.ads = self.register_controller(android_device)
```

- Mobly loads Android devices defined in YAML config (**`wifi_config.yaml`**)
- Connects them via **`adb`**

```
python

self.ad = self.ads[0]
```

- Selects the first Android device

```
python

self.ad.load_snippet(name='wifi', package='com.example.wifisnippet')
```

- Starts the Mobly Snippet APK (**`com.example.wifisnippet`**)
- Opens a gRPC communication pipe
- Now **`self.ad.wifi`** is a **proxy** object
  → calling **`self.ad.wifi.setWifiEnabled(True)`** goes to Java!

✅ 2. Mobly → Calls Test Methods like:

```
python

test_toggle_wifi_and_check_status()
```

Inside:

```
python

self.ad.wifi.setWifiEnabled(True)
```

- Python Mobly calls gRPC → Java method **`@Rpc setWifiEnabled`**
- Inside your snippet APK → Java code calls **`wifiManager.setWifiEnabled(true)`**

Same for:

```
python

self.ad.wifi.isWifiEnabled()
```

And so on.

✅ 3. Mobly → Calls `teardown_class()`

At the end, Mobly framework auto-calls:

```
python

teardown_class()
```

Which does:

```
python

self.ad.wifi.stop_app()
```

That stops the Mobly snippet service cleanly.

🧭 Who Calls Whom — Call Tree

```
text

Mobly Runner
└── WifiControlTest.run()
    ├── setup_class()
    │   ├── register_controller(android_device)
    │   └── load_snippet(name='wifi', ...)
    ├── test_toggle_wifi_and_check_status()
    │   ├── self.ad.wifi.setWifiEnabled(True)    --> Calls Java via gRPC
    │   ├── self.ad.wifi.isWifiEnabled()         --> Calls Java via gRPC
    │   └── ...
    ├── test_get_connected_ssid()
    │   └── self.ad.wifi.getConnectedSSID()      --> Calls Java via gRPC
    └── teardown_class()
        └── self.ad.wifi.stop_app()
```

🔚 Summary

Phase	Function	Who calls it	Purpose
Setup	**`setup_class()`**	Mobly framework	Set up device and snippet
Test	**`test_*()`**	Mobly test runner	Run your actual test cases
Teardown	**`teardown_class()`**	Mobly framework	Clean up resources
Snippet RPC	**`self.ad.wifi.method()`**	gRPC to Java Snippet	Device-side logic

If you want, I can draw this as a **visual flowchart** or explain the **Java side flow** too. Would you like that?

> mobly vs tradefed

Great question! Here's a **clear comparison between Mobly and Tradefed** (also known as Trade Federation), especially in the context of **Android testing** like CTS/VTS or custom device validation.

📊 Mobly vs Tradefed – Overview Table

Feature/Aspect	Mobly	Tradefed (Trade Federation)
🧠 **What it is**	Python-based test framework	Java-based test framework
🛠️ **Primary Language**	Python	Java
📱 **Typical Use Case**	Custom device tests, lab automation, Wi-Fi, etc.	CTS, VTS, GTS, etc. (official Android compliance tests)
🚀 **How to run**	`python3 test.py -c config.yaml`	`cts-tradefed run cts` / `vts-tradefed run vts`
🔌 **Device Setup**	YAML config (Mobly controllers)	XML configs / command-line args
🔍 **Logging**	Mobly-style logging with HTML + JSON reports	Unified test logs, result XML, logs/bugreports
🧩 **Test Type**	Functional, integration, hardware validation	Compatibility, kernel interface, HAL/API compliance
🤝 **Extensibility**	Easy to write Python tests & snippets	Requires writing Java test modules or jars
🔪 **Snippet Support**	Yes (Mobly snippets in APK)	No built-in snippet RPC; uses instrumented APKs
📂 **Framework location**	`tools/mobly/` (or pip install)	AOSP: `tools/tradefederation/`
⚙️ **Official Role**	Internal/custom testing (Google, OEMs)	Official CTS, VTS test runner for Android compliance
🧵 **Multidevice/Parallel**	Yes (via Mobly controller)	Yes (device sharding and modules)

🔍 When to Use Which?

Use Case	Use
Wi-Fi, telephony, Bluetooth custom tests	✅ **Mobly**
Running Android **CTS**, **VTS**, **GTS**	✅ **Tradefed**
Custom Python-based automation	✅ **Mobly**
Java-based app testing, instrumentation	✅ **Tradefed**
Testing multiple DUTs via **adb** + YAML	✅ **Mobly**
Google Play certification / GSI testing	✅ **Tradefed**

⚙️ Architecture Differences

✅ Mobly:

- Written in Python.
- Uses YAML config to connect to devices.

- Supports Android device controller and `snippet` APK for calling Java APIs.
- Simple to extend.
- Great for internal lab/hardware/control tests.

✅ Tradefed:

- Java-based.
- Deeply integrated with AOSP.
- Can run full CTS/VTS suites (e.g., `cts-tradefed run cts`).
- Supports result reporting, retries, sharding, module filters, etc.
- Not easy to modify unless you're familiar with Java + AOSP build system.

🔄 Example Comparison

Mobly Test (`wifi_test.py`)	Tradefed Test (e.g. CTS)
Write in Python using Mobly framework	Write Java test extending `DeviceTestCase`
Load snippet APK to talk to Java code	Use Instrumentation APKs via Tradefed
YAML config: serials, params	XML or CLI config: `--serial`, `--module`
Run with: `python3 wifi_test.py -c conf.yaml`	Run with: `cts-tradefed run cts --module WiFiTest`

🤖 Which One Is Used by Google?

Test Type	Framework Used
CTS, VTS, GTS	✅ **Tradefed**
Pixel hardware tests	✅ **Mobly** (internally)
Network validation	✅ **Mobly**
HAL/API conformance	✅ **Tradefed**

🧠 Summary

- **Mobly**: Python, flexible, easier for lab/device automation.
- **Tradefed**: Java, official runner for Android compliance tests like CTS/VTS.

Would you like a **diagram of how Mobly vs Tradefed execute tests**, or an **example test from both** for comparison?