Cuttlefish

Cuttlefish virtual Android devices

bookmark_border
Cuttlefish is a configurable virtual Android device that can run both remotely (using third-party cloud offerings such as Google Cloud Engine) and locally (on Linux x86 and ARM64 machines).

Cuttlefish goals
Free the platform and app developer from being dependent on physical hardware to develop and validate code changes.
Replicate the framework-based behavior of a real device with a focus on high fidelity by maintaining close alignment with the core framework.
Support all API levels after 28.
Provide a consistent level of functionality across API levels, aligned with the behavior on physical hardware.
Enable scale:

Provide the ability to run multiple devices in parallel.
Enable concurrent test execution with high fidelity at a lower cost of entry.
Provide a configurable device with the ability to adjust form factor, RAM, CPUs, etc.

Compare Cuttlefish to other devices
Cuttlefish and Android Emulator
There are many similarities with the Android Emulator, but Cuttlefish guarantees full fidelity with Android framework (whether this is pure AOSP or a custom implementation in your own tree). In real-world application, this means that you should expect Cuttlefish to respond to your interactions at the OS level just like a physical phone target built with the same customized or pure Android OS source.

The Android Emulator has been built around the use case of making app development easy, and it contains many functional hooks to appeal to the use cases of the Android app developer. This may present challenges if you want to build an emulator with your customized Android framework. If you need a virtual device that will be representative of your custom platform/framework code or tip-of-tree Android, then Cuttlefish is an ideal virtual option. It is the canonical device for representing the current state of AOSP development.

Cuttlefish and physical device
The primary differences between a Cuttlefish virtual device and your physical device are at the hardware abstraction layer (HAL) level, as well as any software that interacts with any custom hardware. Except for hardware-specific implementations, you should expect functionally

equivalent behavior between Cuttlefish and a physical device.

How can Cuttlefish help?
You can interact with Cuttlefish just like you would with any other Android device that you might use for debugging. It will register itself as a normal device via adb and you can interact with it like a physical device via remote desktop. The use cases are broad and can span app testing, custom system build testing and more.

Because Cuttlefish strives for full framework fidelity, it can be used for functional testing of your framework or apps where there are no physical hardware dependencies that are impossible to emulate.

How is Cuttlefish commonly used for testing today?
Some common applications of Cuttlefish for testing include:

CTS
Framework compliance
Continuous integration testing
Custom test suites
Can I host Cuttlefish in the cloud?
Yes, Cuttlefish natively supports Google Cloud and support for other cloud platforms is planned.

Get started
For guidance on creating a Cuttlefish instance based on AOSP, see Use Cuttlefish.

Here's a **simple breakdown** of what Cuttlefish is and how it compares to other Android device options:

🐟 What is Cuttlefish?

Cuttlefish is a **virtual Android device** that:

- Runs **locally** on Linux (x86 or ARM64) or **remotely** (like on Google Cloud).
- Is part of AOSP (Android Open Source Project).
- Emulates a real Android device with **high fidelity** (very close to actual behavior).

🎯 Goals of Cuttlefish

1. **No need for real hardware** – You can test and develop Android code without a phone.
2. **High framework fidelity** – Behaves just like an actual Android device at the software (framework) level.
3. **Wide API support** – Works with **API level 28 and above**.

4. **Highly scalable** – Can run many devices at once (useful for test farms).
5. **Customizable** – Configure RAM, CPU, screen size, and more.

🔍 Comparison

Feature	Cuttlefish	Android Emulator	Physical Device
Fidelity to AOSP	✅ Full (canonical AOSP reference)	❌ Not always (custom behaviors)	✅ Yes (but with hardware variability)
Custom framework testing	✅ Ideal	❌ May break or need workarounds	✅ Yes
Hardware interaction	❌ Limited to virtual	❌ Limited	✅ Full
Use in CI	✅ Excellent (scalable)	✅ Possible but limited	❌ Not scalable
Cloud support	✅ Native (GCE supported)	✅ Possible	❌ No

💻 How Developers Use It

- Run **CTS** (Compatibility Test Suite) for Android certification.
- Test Android framework changes.
- Validate apps on AOSP builds.
- Run tests in **CI/CD pipelines**.
- Build and test **custom ROMs** or Android variants.

☁️ Cloud Use

Cuttlefish works **natively on Google Cloud Engine (GCE)**.

- Ideal for large-scale automated testing.
- Future support for other clouds planned.

🚀 Getting Started

To launch your own Cuttlefish device:

- Check official docs: **Use Cuttlefish**
- You'll need a **Linux machine**, AOSP source, and some setup steps.

🧠 Summary

Cuttlefish is the **go-to virtual Android device for platform development**, CI testing, and system-level validation. If you're working with AOSP or building your own Android stack, it's better suited than the regular emulator.

Would you like a step-by-step guide on **setting up Cuttlefish locally** or on **Google Cloud**?

can oem and vendor use this?

Yes, **OEMs (Original Equipment Manufacturers)** and **Vendors can absolutely use Cuttlefish**, and in fact, **they are encouraged to use it** in many Android platform workflows. Here's how and why:

✅ Why OEMs and Vendors Use Cuttlefish

1. Framework & HAL Development

- Cuttlefish is ideal for testing **AOSP changes** or **custom Android framework layers**.
- OEMs can test platform code **before it reaches physical devices**.

2. CTS/VTS Testing

- OEMs and vendors are required to pass **CTS (Compatibility Test Suite)** and **VTS (Vendor Test Suite)** for certification.
- Cuttlefish supports running these tests **at scale**.
- This avoids the need to flash physical devices for every test cycle.

3. CI/CD Integration

- OEMs integrate Cuttlefish into **continuous integration systems** to:
  - Run nightly or per-commit tests.
  - Validate vendor modules or system images.
  - Catch regressions early in development.

4. Customized Builds

- Cuttlefish allows testing of **custom device configurations**:
  - Different screen sizes, RAM/CPU counts, input/output devices, etc.
- OEMs can simulate devices **without waiting for hardware prototypes**.

5. Remote Testing / Dev Access

- Developers and testers in distributed teams can use Cuttlefish hosted on cloud (e.g., GCP).
- No need to physically distribute devices across teams.

🛠️ What OEMs/Vendors Can Customize in Cuttlefish

Component	Can it be Customized in Cuttlefish?
System image (`system.img`)	✅ Yes
Vendor image (`vendor.img`)	✅ Yes
Kernel	✅ Yes
HALs (via VNDK)	✅ Yes
Boot config	✅ Yes
Device configuration (CPU, screen size, RAM)	✅ Yes

You can build Cuttlefish images using your **own device-specific trees**, including custom HALs and proprietary components, and test them just like on real hardware (with some hardware limitations).

⚠️ Limitations for OEMs/Vendors

Limitation	Reason
No access to real hardware (e.g., camera sensors, fingerprint)	Virtualization limit
Some hardware-specific HALs may fail	If tightly coupled with physical hardware
Performance may differ from real devices	Especially for graphics-intensive tasks

But for anything that runs at the **Android framework, system, or app level**, Cuttlefish is **very close to real behavior**.

✅ Conclusion

Yes, **OEMs and vendors can and do use Cuttlefish**, especially for:

- Early development
- Framework and system integration testing
- CTS/VTS validation
- CI automation

Would you like an **OEM-specific workflow** for integrating Cuttlefish into your Android build and test pipeline?

Get started

bookmark_border
This page describes how to launch an AOSP build using Cuttlefish.

Verify KVM availability
Cuttlefish is a virtual device and is dependent on virtualization being available on the host machine.

In a terminal on your host machine, make sure that virtualization with a Kernel-based Virtual Machine (KVM) is available:

```
grep -c -w "vmx\|svm" /proc/cpuinfo
```

This command should return a nonzero value.

Note: On a cloud machine, you might need to follow cloud-specific or vendor-specific steps to enable KVM. For Google Compute Engine (GCE), see About nested virtualization.
When running on an ARM64 machine, the most direct way is to check for /dev/kvm:

```
find /dev -name kvm
```

Tip: You can use this command to confirm support of KVM on any environment.
Launch Cuttlefish
In a terminal window, download, build, and install the host Debian packages:

```
sudo apt install -y git devscripts equivs config-package-dev debhelper-compat golang curl
git clone https://github.com/google/android-cuttlefish
cd android-cuttlefish
tools/buildutils/build_packages.sh
sudo dpkg -i ./cuttlefish-base_*_*64.deb || sudo apt-get install -f
sudo dpkg -i ./cuttlefish-user_*_*64.deb || sudo apt-get install -f
sudo usermod -aG kvm,cvdnetwork,render $USER
sudo reboot
```

The reboot triggers installing additional kernel modules and applies udev rules.

Cuttlefish is part of the Android Open-Source Platform (AOSP). Builds of the virtual device are found at the Android Continuous Integration site. To find an index of all Android builds, navigate to the Android Continuous Integration site at http://ci.android.com/.

Enter a branch name. Use the default aosp-android-latest-release branch or use a generic system image (GSI) branch such as aosp-android13-gsi.

Navigate to the aosp_cf_x86_64_only_phone build target and click userdebug for the latest build.

Tip: For ARM64, use the branch aosp-android-latest-release and the device target aosp_cf_arm64_only_phone-userdebug.
Click the green box below userdebug to select this build. A Details panel appears with more information specific to this build. In this panel, click Artifacts to see a list of all the artifacts attached to this build.

In the Artifacts panel, download the artifacts for Cuttlefish.

Click the aosp_cf_x86_64_phone-img-xxxxxx.zip artifact for x86_64 or the aosp_cf_arm64_only_phone-xxxxxx.zip artifact for ARM64, which contains the device images. In the filename, "xxxxxx" is the build ID for this device.

Scroll down in the panel and download cvd-host_package.tar.gz. Always download the host package from the same build as your images.

On your local system, create a container folder and extract the packages:

x86_64 architecture:

```
mkdir cf
cd cf
tar -xvf /path/to/cvd-host_package.tar.gz
unzip /path/to/aosp_cf_x86_64_phone-img-xxxxxx.zip
```

ARM64 architecture:

```
mkdir cf
cd cf
tar -xvf /path/to/cvd-host_package.tar.gz
unzip /path/to/aosp_cf_arm64_only_phone-img-xxxxxx.zip
```

Launch Cuttlefish:

```
HOME=$PWD ./bin/launch_cvd --daemon
```

Verify that Cuttlefish is visible through adb
Similar to a physical device, Cuttlefish is visible through the Android Debug Bridge (adb).

In the same folder where you launched Cuttlefish, run the following command to see a list of all the Android devices available through adb on your host machine:

```
./bin/adb devices
```

View and interact with the virtual device in the web

By default, Cuttlefish launches with --start_webrtc, which enables a webview through port 8443 on the host machine.

To view and interact with your virtual devices, navigate to https://localhost:8443 in your web browser.

For more information, see Cuttlefish: WebRTC Streaming.

Stop Cuttlefish
Stop the virtual device within the same directory as you used to launch the device:

HOME=$PWD ./bin/stop_cvd

Here's a **step-by-step simplified guide** for setting up and running **Cuttlefish** virtual Android devices locally from AOSP:

🛠️ Prerequisites

Before you begin:

- You need a **Linux x86_64 or ARM64 machine**.
- **Virtualization (KVM)** must be supported and enabled.

✅ Step 1: Verify KVM Availability

Open a terminal and run:

For x86_64:

```bash
grep -c -w "vmx\|svm" /proc/cpuinfo
```

- Output must be **non-zero** → means virtualization is supported.

For ARM64:

```bash
find /dev -name kvm
```

- If **`/dev/kvm`** exists, KVM is enabled.

If you're using a **cloud VM** (e.g., GCE), ensure **nested virtualization** is enabled.

📦 Step 2: Install Build Tools

```bash
sudo apt update
sudo apt install -y git devscripts equivs config-package-dev debhelper-compat golang curl
```

🧱 Step 3: Build and Install Cuttlefish Host Packages

```bash
git clone https://github.com/google/android-cuttlefish
cd android-cuttlefish
tools/buildutils/build_packages.sh
```

Install the generated `.deb` packages:

```bash
sudo dpkg -i ./cuttlefish-base_*_*64.deb || sudo apt-get install -f
sudo dpkg -i ./cuttlefish-user_*_*64.deb || sudo apt-get install -f
```

Add user to necessary groups:

```bash
sudo usermod -aG kvm,cvdnetwork,render $USER
sudo reboot
```

🌐 Step 4: Download AOSP Cuttlefish System Images

1. Go to: http://ci.android.com/
2. Choose branch:
    - `aosp-android-latest-release` (recommended)
    - Or GSI branch like `aosp-android13-gsi`
3. Select build target:
    - For x86: `aosp_cf_x86_64_only_phone-userdebug`
    - For ARM64: `aosp_cf_arm64_only_phone-userdebug`

4. Click the green **userdebug** box → Go to **Artifacts**
5. Download:
   - `aosp_cf_x86_64_phone-img-*.zip` (or ARM64 variant)
   - `cvd-host_package.tar.gz` (from same build)

📂 Step 5: Extract Artifacts

```bash
mkdir cf
cd cf
tar -xvf /path/to/cvd-host_package.tar.gz
unzip /path/to/aosp_cf_x86_64_phone-img-xxxxxx.zip
```

(Replace paths with actual downloaded file paths.)

🚀 Step 6: Launch the Virtual Device

```bash
HOME=$PWD ./bin/launch_cvd --daemon
```

🔌 Step 7: Verify Device with ADB

```bash
./bin/adb devices
```

You should see a device like:

```makefile
localhost:6520   device
```

🌍 Step 8: Interact via Web Interface

Open your browser:

```arduino
https://localhost:8443
```

This opens a full virtual Android UI via WebRTC.

> Accept the browser's SSL warning if needed.

🛑 Step 9: Stop Cuttlefish

From the same directory:

```bash
HOME=$PWD ./bin/stop_cvd
```

🧠 Summary

Cuttlefish is now running on your local machine like a real Android phone, complete with:

- ADB access
- Web UI
- Userdebug capabilities
- Custom AOSP images

Would you like a script or automation setup (e.g., a Makefile or bash script) for all of this?

confusing explain in simple words

Sure! Let me explain in **very simple steps** what to do to run a **Cuttlefish virtual Android phone** on your Linux machine.

🧾 What is Cuttlefish?

Cuttlefish is like a **virtual Android phone** that runs on your **Linux computer**, without needing a real device.

✅ Step 1: Check if your computer supports virtualization (KVM)

Open Terminal and run:

```bash
grep -c -w "vmx\|svm" /proc/cpuinfo
```

- If the result is **not 0**, then it's okay.
  (Your CPU supports KVM = Virtual Android will work.)

🧰 Step 2: Install the required tools

In Terminal, run:

```bash
sudo apt update
sudo apt install -y git curl golang devscripts equivs config-package-dev debhelper-compat
```

This installs tools needed to build and run Cuttlefish.

📥 Step 3: Get the Cuttlefish source and build it

Run these commands one by one:

```bash
git clone https://github.com/google/android-cuttlefish
cd android-cuttlefish
tools/buildutils/build_packages.sh
```

📦 Step 4: Install Cuttlefish software

Now install the built `.deb` packages:

```bash
sudo dpkg -i ./cuttlefish-base_*.deb || sudo apt-get install -f
sudo dpkg -i ./cuttlefish-user_*.deb || sudo apt-get install -f
```

👤 Step 5: Give your user access to required permissions

```bash
sudo usermod -aG kvm,cvdnetwork,render $USER
```

Then restart your computer:

```bash
sudo reboot
```

🌐 Step 6: Download Android virtual phone image

1. Go to: https://ci.android.com/
2. Choose branch: **`aosp-android-latest-release`**
3. Click **`aosp_cf_x86_64_only_phone-userdebug`**
4. Click green **userdebug** box.
5. Download 2 files:
    - One ending with **`.zip`** (this is the Android image)
    - One called **`cvd-host_package.tar.gz`** (host tools)

📂 Step 7: Extract the files

Create a folder and extract:

```bash
mkdir cf
cd cf
tar -xvf /path/to/cvd-host_package.tar.gz
unzip /path/to/aosp_cf_x86_64_phone-img-xxxxxx.zip
```

Replace **`/path/to/...`** with your real file paths.

🚀 Step 8: Start the Android virtual phone

```bash
HOME=$PWD ./bin/launch_cvd --daemon
```

This will start the Android phone in the background.

🔍 Step 9: Check if it is working

Run:

```bash
./bin/adb devices
```

If it works, you'll see a line like:

```makefile
localhost:6520  device
```

🌍 Step 10: See the Android phone in your browser

Open:

```arduino
https://localhost:8443
```

You'll see the Android screen. Click and use it like a real phone.

🛑 To stop the virtual phone

Run:

```bash
HOME=$PWD ./bin/stop_cvd
```

Would you like me to give you one **simple script** that does all of this?

Cuttlefish: Snapshot and restore

bookmark_border
Android 15 introduces how to take a snapshot and restore a snapshot of a Cuttlefish virtual device. Taking a snapshot of a Cuttlefish device lets you save the state of the device in an image on disk. You can then restore the snapshot to bring up a Cuttlefish device to the previously

saved state.

You can use snapshots in various automated or manual workflows. When performing a procedure that modifies the device, you can take a snapshot of the device to ensure you can restore the device back to a specified state. For example, when running test suites that modify the state of the device and can potentially cause issues or instabilities in the following tests, you can take a snapshot so you can restore the device to a saved state after a test, ensuring that subsequent tests run cleanly.

Another example workflow where snapshots are useful is when testing the behavior of an app. When testing the behavior of an app based on a set of actions, you can take a snapshot in between actions as the app is running to let you restore to that snapshot without having to restart from the beginning. For example, if launching a game has a long bringup time, you can take a snapshot after reaching the main menu so you can restore the device to that state, skipping the bringup time.

Take snapshot of a Cuttlefish device
Note: Snapshots are only supported on x86_64 platforms.
When taking a snapshot of a device, the Cuttlefish device must be suspended to make sure the Cuttlefish device is in a stable state. When the device is suspended, all vCPUs and devices are stopped, and all buffers push their state to the VM. The snapshot then saves the vCPU state, memory, and device state to disk at a specified destination folder.

VirtiosFS isn't supported and must be disabled when taking a snapshot. To disable VirtioFS, pass the argument --enable_virtiofs=false when running cvd create or cvd start.

Only the SwiftShader (guest_swiftshader) GPU mode is supported for snapshots. Other accelerated graphics modes aren't supported.

The following steps describes the process of launching a Cuttlefish device and taking a snapshot.

Launch a device, while disabling VirtioFS. (You can then use the device.)

cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader
Take a snapshot by running cvd snapshot_take with the following flags:

--force: If a folder exists at the specified snapshot path, this flag ensures that the existing folder is deleted, and a new folder is created at the snapshot path containing the snapshot

--auto_suspend: Suspends the device before the snapshot is taken and resumes the device after the snapshot is taken.

--snapshot_path: The specified path where a new folder is created with the snapshot.

cvd snapshot_take --force --auto_suspend \
--snapshot_path=PATH
Restore a Cuttlefish device
When restoring a Cuttlefish device snapshot, the Cuttlefish instance for which the snapshot was taken must be stopped. If the instance is already stopped, no further action is required and the snapshot can be restored.

To restore a snapshot of a Cuttlefish device, launch a device with cvd create and include the snapshot path. If the base instance number of the Cuttlefish device on which the snapshot was taken is different from the base instance number of the current Cuttlefish instance, pass that base instance number using the --base_instance_num flag.

cvd create --snapshot_path=PATH \
--base_instance_num=ID
Suspend a Cuttlefish device
You can suspend a Cuttlefish device without taking a snapshot (no disk space is used for saving the state). To suspend a Cuttlefish device, run:

cvd suspend
Resume a Cuttlefish device
To resume a suspended Cuttlefish device, run:

cvd resume
Validate the Snapshot/Restore feature
The Snapshot/Restore feature can be validated by running the following test:

atest SnapshotTest

Here's a **simple explanation** of the **Snapshot and Restore** feature for **Cuttlefish (Android 15+)**, step by step:

🧠 What is Snapshot and Restore?

Cuttlefish now supports saving and loading the **exact state** of a virtual Android phone — like **saving a game** and **loading it later**.

📸 What is a Snapshot?

A **snapshot**:

- Saves everything: memory, CPU state, device state.
- Lets you **go back** to that saved state anytime.
- Helps in testing or debugging.

✅ Use cases:

- Run tests → Snapshot before test → Restore after test.
- Skip setup steps (like game loading) by restoring to a ready state.

🛠️ Requirements

- ✅ Only works on **x86_64** systems (not ARM64 yet).
- ❌ **VirtioFS** must be **disabled**.
- ✅ Use **guest_swiftshader** GPU mode (software rendering only).

🚀 Steps to Use Snapshot and Restore

✅ Step 1: Launch Cuttlefish with snapshot support

Run this command to create the device with correct options:

```bash
cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader
```

This prepares the virtual phone with snapshot compatibility.

📸 Step 2: Take a snapshot

Now run:

```bash
cvd snapshot_take --force --auto_suspend --snapshot_path=/path/to/my_snapshot
```

🔹 **`--force`** → Deletes existing snapshot folder if it exists
🔹 **`--auto_suspend`** → Freezes the device before saving, then resumes
🔹 **`--snapshot_path=...`** → Where to save the snapshot on disk

🔁 Step 3: Restore from snapshot

First, stop the running device:

```bash
cvd stop
```

Then restore using:

```bash
cvd create --snapshot_path=/path/to/my_snapshot
```

> If you're restoring to a different instance ID (not 1), add:

```bash
--base_instance_num=2
```

💤 Step 4: Suspend / Resume manually (optional)

- Suspend (pause the VM, but don’t save to disk):

```bash
cvd suspend
```

- Resume (continue running):

```bash
cvd resume
```

✅ Step 5: Test if it works

To verify that snapshot/restore is working correctly, run:

```bash
atest SnapshotTest
```

This runs an automated test to confirm everything behaves correctly.

🧠 Summary

Action	Command Example
Launch device with snapshot support	`cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader`
Take snapshot	`cvd snapshot_take --force --auto_suspend --snapshot_path=PATH`
Restore snapshot	`cvd create --snapshot_path=PATH`
Suspend device	`cvd suspend`
Resume device	`cvd resume`
Run test	`atest SnapshotTest`

Would you like a **simple shell script** that automates snapshot, restore, and suspend/resume steps for you?

Cuttlefish: Snapshot and restore

bookmark_border
Android 15 introduces how to take a snapshot and restore a snapshot of a Cuttlefish virtual device. Taking a snapshot of a Cuttlefish device lets you save the state of the device in an image on disk. You can then restore the snapshot to bring up a Cuttlefish device to the previously saved state.

You can use snapshots in various automated or manual workflows. When performing a procedure that modifies the device, you can take a snapshot of the device to ensure you can restore the device back to a specified state. For example, when running test suites that modify the state of the device and can potentially cause issues or instabilities in the following tests, you can take a snapshot so you can restore the device to a saved state after a test, ensuring that subsequent tests run cleanly.

Another example workflow where snapshots are useful is when testing the behavior of an app. When testing the behavior of an app based on a set of actions, you can take a snapshot in between actions as the app is running to let you restore to that snapshot without having to restart from the beginning. For example, if launching a game has a long bringup time, you can take a snapshot after reaching the main menu so you can restore the device to that state, skipping the bringup time.

Take snapshot of a Cuttlefish device
Note: Snapshots are only supported on x86_64 platforms.
When taking a snapshot of a device, the Cuttlefish device must be suspended to make sure the Cuttlefish device is in a stable state. When the device is suspended, all vCPUs and devices are

stopped, and all buffers push their state to the VM. The snapshot then saves the vCPU state, memory, and device state to disk at a specified destination folder.

VirtiosFS isn't supported and must be disabled when taking a snapshot. To disable VirtioFS, pass the argument --enable_virtiofs=false when running cvd create or cvd start.

Only the SwiftShader (guest_swiftshader) GPU mode is supported for snapshots. Other accelerated graphics modes aren't supported.

The following steps describes the process of launching a Cuttlefish device and taking a snapshot.

Launch a device, while disabling VirtioFS. (You can then use the device.)

```
cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader
```

Take a snapshot by running cvd snapshot_take with the following flags:

--force: If a folder exists at the specified snapshot path, this flag ensures that the existing folder is deleted, and a new folder is created at the snapshot path containing the snapshot

--auto_suspend: Suspends the device before the snapshot is taken and resumes the device after the snapshot is taken.

--snapshot_path: The specified path where a new folder is created with the snapshot.

```
cvd snapshot_take --force --auto_suspend \
--snapshot_path=PATH
```

Restore a Cuttlefish device

When restoring a Cuttlefish device snapshot, the Cuttlefish instance for which the snapshot was taken must be stopped. If the instance is already stopped, no further action is required and the snapshot can be restored.

To restore a snapshot of a Cuttlefish device, launch a device with cvd create and include the snapshot path. If the base instance number of the Cuttlefish device on which the snapshot was taken is different from the base instance number of the current Cuttlefish instance, pass that base instance number using the --base_instance_num flag.

```
cvd create --snapshot_path=PATH \
--base_instance_num=ID
```

Suspend a Cuttlefish device

You can suspend a Cuttlefish device without taking a snapshot (no disk space is used for saving the state). To suspend a Cuttlefish device, run:

cvd suspend
Resume a Cuttlefish device
To resume a suspended Cuttlefish device, run:

cvd resume
Validate the Snapshot/Restore feature
The Snapshot/Restore feature can be validated by running the following test:

atest SnapshotTest

Here's a **very simple explanation** of how the **Cuttlefish Snapshot & Restore** feature works (available in **Android 15+**):

🧠 What is it?

Snapshot = Save the entire state of a virtual Android phone (Cuttlefish).
Restore = Go back to that saved state instantly — like a save/load in a game.

✅ Why is this useful?

- 🧪 **Test Reset**: After a test changes the device, you can restore to a clean state.
- 🕹️ **Skip Long Setup**: If a game takes long to load, you can save at the main menu and skip next time.
- 🔁 **Repeat Tests**: Easily re-run tests from the same app or system state.

🚦 Snapshot Requirements (IMPORTANT)

Requirement	Value
CPU Architecture	Only works on **x86_64**
GPU mode	Must use **`--gpu_mode=guest_swiftshader`**
VirtioFS (file sharing)	Must be **disabled** using **`--enable_virtiofs=false`**

🧪 How to Use Snapshot and Restore

🔧 1. Create the virtual device (with correct options)

```bash
cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader
```

This starts a virtual Android phone with snapshot support.

📸 2. Take a Snapshot

```bash
cvd snapshot_take --force --auto_suspend --snapshot_path=/path/to/my_snapshot
```

✅ What it does:

- **`--force`**: Overwrites old snapshot if any.
- **`--auto_suspend`**: Freezes the device, takes snapshot, then resumes it.
- **`--snapshot_path`**: Where the snapshot will be saved.

🛑 3. Stop the Device Before Restore

```bash
cvd stop
```

🔁 4. Restore from Snapshot

```bash
cvd create --snapshot_path=/path/to/my_snapshot
```

If you're using a different instance ID (not default 1), use:

```bash
cvd create --snapshot_path=/path/to/my_snapshot --base_instance_num=2
```

😴 5. Suspend/Resume (optional)

You can pause/resume the device manually (without saving):

- Suspend:

```bash
cvd suspend
```

- Resume:

```bash
cvd resume
```

🧪 6. Test if snapshot/restore works

Run this test:

```bash
atest SnapshotTest
```

🧠 Summary Table

Action	Command
Create device	`cvd create --enable_virtiofs=false --gpu_mode=guest_swiftshader`
Take snapshot	`cvd snapshot_take --force --auto_suspend --snapshot_path=PATH`
Stop device	`cvd stop`
Restore snapshot	`cvd create --snapshot_path=PATH`
Suspend device	`cvd suspend`
Resume device	`cvd resume`
Test snapshot	`atest SnapshotTest`

Would you like a **single shell script** that automates snapshot and restore for you?

Cuttlefish: GPU graphics acceleration

bookmark_border
Cuttlefish's accelerated graphics mode uses your host machine's physical graphics processing unit (GPU) for rendering by passing guest rendering commands to your host machine, running the rendering commands calls on your host machine, and passing the rendered results back to the guest.

A Cuttlefish device running in Android 11 or higher detects and uses accelerated graphics. If the host machine doesn't support accelerated graphics or the Android version is Android 10 or lower, guest-side rendering (for example, UI and video playback) in your Cuttlefish device is handled by SwiftShader. SwiftShader is a software implementation of the OpenGL and Vulkan APIs. Because SwiftShader is a software implementation, it provides a universally accessible rendering solution for Cuttlefish capable of running on any host machine.

However, using SwiftShader isn't as performant as a normal device. Rendering is a parallel problem that can be massively parallelized, as pixel values can be computed independently. Graphics processing units (GPUs) are hardware units that address this problem by accelerating rendering.

Requirements
Accelerated graphics mode requires that the host have:

EGL capable driver supporting the GL_KHR_surfaceless_context extension
OpenGL ES capable driver
Vulkan capable driver
Use accelerated graphics modes
GfxStream
To use the GfxStream accelerated graphics mode, launch your local Cuttlefish device with the --gpu_mode=gfxstream flag. Using this mode, OpenGL and Vulkan API calls are forwarded directly to the host.

launch_cvd --gpu_mode=gfxstream
Virgl
To use the Virgl accelerated graphics mode, launch your local Cuttlefish device with the --gpu_mode=drm_virgl flag.

launch_cvd --gpu_mode=drm_virgl
When using the Virgl accelerated graphics mode, OpenGL API calls are translated into an intermediate representation (see Gallium3D). The intermediate representation is communicated to the host and the virglrenderer library on the host translates the intermediate representation back to OpenGL API calls.

Note: Vulkan isn't supported.

Here's a **very simple explanation** of how **GPU graphics acceleration** works in **Cuttlefish**, and what the options mean:

🧠 What's the problem?

Normally, **drawing graphics (UI, videos, animations)** needs a **GPU**.

But Cuttlefish (Android emulator) runs inside your PC, and:

- If it **can't use your PC's GPU**, it uses **SwiftShader** (a very slow software renderer).
- If it **can use your GPU**, it will be much **faster and smoother**.

🚀 Accelerated Graphics = Use Host GPU

When enabled, Cuttlefish lets Android send drawing commands to your **PC's real GPU** to render faster.

This is called **GPU acceleration**.

✅ Requirements

To enable GPU acceleration, your **PC** must support:

Feature	What it means
EGL + **`GL_KHR_surfaceless_context`**	Modern OpenGL environment (for rendering without a window)
OpenGL ES	GPU support for mobile-style graphics
Vulkan	Optional, but helps Vulkan-based apps

🧪 GPU Modes in Cuttlefish

You can choose **how** Cuttlefish uses the GPU using the **`--gpu_mode`** flag when launching.

---

1. 🎮 `gfxstream` (Best, Recommended)

```bash
launch_cvd --gpu_mode=gfxstream
```

- 🏎️ **Fastest**
- 💻 Sends **OpenGL + Vulkan** commands directly to your PC's GPU.
- ✅ Recommended for **modern systems**
- 🧪 Works with Android 11+

---

2. 🎨 `drm_virgl` (Good for OpenGL only)

```bash
launch_cvd --gpu_mode=drm_virgl
```

- 🧱 Uses **Virgl + Gallium3D**
- 🔄 Converts graphics commands to an intermediate format
- ✅ Supports **OpenGL**
- ❌ **No Vulkan support**
- ⚠️ Slower than **`gfxstream`**, but still faster than software rendering

---

3. 🐢 `guest_swiftshader` (Fallback, slowest)

```bash
launch_cvd --gpu_mode=guest_swiftshader
```

- 🧠 Pure software renderer (no GPU)
- 🐌 Very slow, only for compatibility
- ✅ Works everywhere, even with bad drivers

💡 Summary

Mode	Vulkan Support	Speed	Use When...
`gfxstream`	✅ Yes	🟢 Fastest	Your PC has good OpenGL/Vulkan support
`drm_virgl`	❌ No	🟡 Medium	You only need OpenGL
`guest_swiftshader`	❌ No	🔴 Very Slow	Nothing else works

🚀 Example: Start Cuttlefish with GPU Acceleration

```bash
launch_cvd --gpu_mode=gfxstream
```

✅ That will give you the **fastest, real-GPU-accelerated Cuttlefish UI** (if your host supports it).

Would you like help checking if your **host system supports `gfxstream`** mode or EGL/Vulkan?

Cuttlefish control panel

bookmark_border
The default WebRTC browser interface for Cuttlefish includes a control panel that enables more ways to interact with the virtual device.

The control panel features default buttons to simulate common physical device actions such as power button or volume buttons, as well as device rotation.

Custom actions
You can customize the control panel to add more buttons that allow your virtual device to more closely emulate your physical device. This is useful for testing features unique to your device, such as a hardware button or special gesture that triggers a unique action in the OS. You can also use custom buttons to enable testing more QA-focused features such as the behavior of your OS when the device is low battery.

The default Cuttlefish control panel includes support to "plug in" custom actions without needing to modify the main Cuttlefish AOSP project. Your virtual device needs to include only a minimal configuration file to start using custom actions. See this example custom action config file.

Create a JSON file that defines your device's custom actions. You can put this file in any directory you own. The structure of this file is described in the ADB shell and Action server sections.

Create a prebuilt_etc_host module for your JSON config. Ensure that the sub_dir is equal to cvd_custom_action_config.

```
prebuilt_etc_host {
    // Use any name you choose.
    name: "my_custom_action_config.json",
    src: "my_custom_action_config.json",
    // Always use this sub_dir.
    sub_dir: "cvd_custom_action_config",
}
```

Set Soong config build variables in your device's product makefile to configure the virtual device host package to include your custom action config file.

```
# Set these variables exactly as shown here to enable the host package to see
# your custom config module name.
SOONG_CONFIG_NAMESPACES += cvd
SOONG_CONFIG_cvd += custom_action_config

# Set this value to the name of your JSON module.
```

SOONG_CONFIG_cvd_custom_action_config := my_custom_action_config.json
Note: If you want to remove custom actions from a local build, or change the name of the custom action config module, remove your old config by deleting ${ANDROID_SOONG_HOST_OUT}/etc/cvd_custom_action_config.
There are two supported methods to implement a custom action:

ADB shell command
Action server
Your JSON config file can define multiple instances of each type of implementation.

ADB shell command
You can define a single button that is implemented by executing a single adb shell command.
For example, the following JSON snippet defines a single button that launches a web page:

```
{
  "shell_command":"am start -a android.intent.action.VIEW -d https://www.android.com/",
  "button":{
     "command":"web",
     "title":"Web Page",
     "icon_name":"language"
  }
}
```

The fields are:

shell_command: The command to execute in adb shell when the button is pressed
button: A single button object with the following subfields:
command: A unique name for this button
title: An alt-text title for this button
icon_name: The name of an icon from https://material.io/resources/icons
Action server
Action servers allow more control over the behavior of your actions. An action server is a host binary that listens for button press events from WebRTC using a socket pair. WebRTC forwards the events to the action server, and then the action server decides how to implement the action.

Action servers allow more powerful control, such as maintaining state (such as for a toggleable event) or even running "meta-actions" such as killing the current device, launching more devices, or starting a screen-recording browser extension. The possibilities are limited only by what you decide to implement inside the host binary.

The following JSON snippet defines an action server that listens for events on two buttons:

```
{
  "server":"cuttlefish_example_action_server",
  "buttons":[
```

```
    {
      "command":"settings",
      "title":"Quick Settings",
      "icon_name":"settings"
    },
    {
      "command":"alert",
      "title":"Do Not Disturb",
      "icon_name":"notifications_paused"
    }
  ]
}
The fields are:

server: The name of your host binary module
buttons: An array of buttons, with the same subfields as above
After updating the JSON config, append the name of the action server module to the Soong
config build variable cvd_custom_action_servers. For example:


# Append to this variable exactly as shown here.
SOONG_CONFIG_cvd += custom_action_servers

# Append the name of your action server(s) to this variable.
SOONG_CONFIG_cvd_custom_action_servers += cuttlefish_example_action_server
Each action server host binary should perform the following steps:

Accept a socket file descriptor number as the first and only program argument.

This socket is created by launch_cvd using socketpair with domain AF_LOCAL, type
SOCK_STREAM, and protocol 0.

In a loop, attempt to read 128 bytes from the socket. These bytes contain button press events
sent by the WebRTC client in the format command:state. command is as provided in the JSON
config, and state is the button press state (down or up).

Act on the incoming events to simulate the custom action.

Warning: You should implement any action servers in projects that you own, not in the main
AOSP Cuttlefish project. This prevents conflicts with future AOSP changes in that project.
```

Here's a **simple explanation** of the **Cuttlefish Control Panel custom actions** feature and whether **OEMs and vendors can use it**:

🔧 What is the Cuttlefish Control Panel?

It's a **web-based interface** used to interact with Android virtual devices (AVDs) running on **Cuttlefish**. It includes buttons like:

- Power button
- Volume up/down
- Rotate screen

🧩 What are custom actions?

You can **add your own buttons** to simulate things like:

- A special hardware button (e.g., "Camera Shutter")
- A gesture (e.g., "Three-finger swipe")
- Test actions (e.g., "Simulate Low Battery")

This helps **OEMs and vendors** test **device-specific features** on virtual devices before actual hardware is ready.

✅ Can OEMs and vendors use this?

Yes! OEMs and vendors can and should use this to test their **custom device behaviors** in virtual environments.

They just need to:

1. Write a **JSON config file** describing the custom buttons.
2. Build that file into the system using a **Soong module**.
3. Tell the build system to include it using **`SOONG_CONFIG_*` variables**.

No need to modify Cuttlefish itself. You keep it separate.

🧪 Two types of custom actions:

1. ADB Shell Command

- Run a shell command when the button is pressed.
- ✅ Very easy.
- Example: Open a webpage with:

```json
{
  "shell_command":"am start -a android.intent.action.VIEW -d https://www.android.com/",
  "button":{
    "command":"web",
    "title":"Web Page",
    "icon_name":"language"
  }
}
```

2. Action Server

- You write your own **host program** to handle button events.
- ✅ More powerful: can toggle things, track state, or run complex logic.
- Your app listens on a socket and gets messages like `settings:down`.

🔒 Important Notes for OEMs/Vendors

- Keep your custom buttons and action servers in **your own projects** (not AOSP/Cuttlefish) to avoid breaking with future AOSP updates.
- You can test your full UI/UX behavior in Cuttlefish **before hardware exists**.

💡 TL;DR:

Feature	Explanation
Control Panel	Web interface for virtual device buttons
Custom Actions	Let you add new buttons
OEM/Vendor Usage	✅ Yes, for testing your unique features
Two methods	Shell command (simple), Action server (powerful)

Want a working example or guide on setting it up for your device?

Cuttlefish: WebRTC streaming

bookmark_border
WebRTC streaming allows users to remotely control their Cuttlefish virtual devices from their browsers, without having to install any other software in the client machine. Other advantages of WebRTC streaming are:

More efficient encoding than VNC

In-browser ADB
Extensible protocol (camera stream, microphone, sensor data are all possible over WebRTC)
Use WebRTC
To use WebRTC and interact with your Cuttlefish device in your browser, do the following:

To launch your devices using WebRTC, add the --start_webrtc=true flag to your launch_cvd invocation.

launch_cvd --start_webrtc=true
To see a list of all the devices you can connect to, point your browser to <https://localhost:8443>.

Port usage
Besides TCP:8443, WebRTC uses other ports to establish a connection and run. These ports must be allowed on the firewall if connecting from a different machine than where Cuttlefish is being executed. The list of ports needed are:

TCP:15550..15599
UDP:15550..15599

Let's break this down into **simple terms**:

🌐 What is WebRTC streaming in Cuttlefish?

WebRTC lets you **see and control a virtual Android device directly in your browser**, without needing any extra software like VNC or Remote Desktop.

✅ Benefits:

Feature	Simple Explanation
✅ More efficient	Better video quality than VNC, less lag
✅ No install needed	Just use Chrome or Firefox
✅ In-browser ADB	You can run ADB commands in the browser
✅ Extensible	You can send/receive camera, mic, sensor data over the stream

---

🚀 How to use it:

1. **Launch Cuttlefish with WebRTC enabled**
   Use this command:

```bash
launch_cvd --start_webrtc=true
```

2. **Open it in your browser**
   Go to:

   ```arduino
   https://localhost:8443
   ```

   You'll see the Cuttlefish UI to interact with your devices.

🔐 Port requirements:

If you're running this on a **remote server** (not your local machine), you need to **allow these ports in the firewall**:

Protocol	Port Range	Used For
TCP	8443	WebRTC control panel (browser)
TCP	15550–15599	WebRTC data and signaling
UDP	15550–15599	WebRTC video/audio streaming

💡 TL;DR:

- You can **stream and control virtual Android devices** in-browser with **great performance**
- Just run **`launch_cvd --start_webrtc=true`**
- Then open **`https://localhost:8443`**
- Open ports **8443, TCP/UDP 15550–15599** if remote

Would you like an example of using WebRTC + Control Panel together?

Cuttlefish: Run stable CTS

bookmark_border
This page describes how to run the Compatibility Test Suite (CTS) to evaluate your virtual device based on the AOSP source code.

Build Cuttlefish
Before running CTS, build the virtual device that you want to test with CTS. For AOSP Cuttlefish,

do the following:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug
make -j
```

Build CTS
CTS is built separately from your device artifacts. To build CTS, follow these instructions.

From the root of the same tree where Cuttlefish was built, run the following commands:

```
source build/envsetup.sh
m -j cts WITH_DEXPREOPT_BOOT_IMG_AND_SYSTEM_SERVER_ONLY=false
TARGET_PRODUCT=aosp_x86_64
```

Launch multi-tenant Cuttlefish instances
Note: For detailed instructions on launching multiple Cuttlefish devices, see Cuttlefish Multi-Tenancy and Powerwashing.
From the root of the same tree where Cuttlefish was built, run the following commands:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug
launch_cvd --num_instances=8 --daemon
```

When the instances start, verify each of the devices is adb accessible.

```
adb devices
```

The output should look similar to the example output below:

```
List of devices attached
127.0.0.1:6520  device
127.0.0.1:6521  device
127.0.0.1:6522  device
127.0.0.1:6523  device
127.0.0.1:6524  device
127.0.0.1:6525  device
127.0.0.1:6526  device
127.0.0.1:6527  device
```

Enable Wi-Fi on Cuttlefish instances
Some of the CTS tests fail if Wi-Fi isn't enabled. To enable Wi-Fi on all locally hosted Cuttlefish devices, follow these instructions.

From the root of the same tree where Cuttlefish was built, run the following commands:

for SERIAL in $(adb devices | grep -E '127.+device$' | cut -f1); do
adb -s ${SERIAL} install -r -g tools/tradefederation/core/res/apks/wifiutil/WifiUtil.apk
adb -s ${SERIAL} shell su root svc wifi enable
adb -s ${SERIAL} shell am instrument -e method "connectToNetwork" -e scan_ssid "false" -e ssid "VirtWifi" -w com.android.tradefed.utils.wifi/.WifiUtil
done
Run CTS stable subset
Not all of the CTS pass on Cuttlefish, as some of the tests are more tailored towards physical devices. To run the CTS with the cts-virtual-device-stable test plan that tracks the latest of what is stable on the virtual platform, follow these instructions.

From the root of the same tree where Cuttlefish was built, run the following commands:

source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug
cts-tradefed run cts-virtual-device-stable --no-enable-parameterized-modules --max-testcase-run-count 2 --retry-strategy RETRY_ANY_FAILURE --reboot-at-last-retry --shard-count 8
Expect the test to run for 20 to 25 minutes, and print out a summary similar to following:

=============== Summary ===============
Total Run time: 22m 8s
184/184 modules completed
Total Tests : 92483
PASSED : 92363
FAILED : 0
IGNORED : 71
ASSUMPTION_FAILURE: 49
Shard 0 used: [127.0.0.1:6522]
Shard 1 used: [127.0.0.1:6525]
Shard 2 used: [127.0.0.1:6524]
Shard 3 used: [127.0.0.1:6521]
Shard 4 used: [127.0.0.1:6520]
Shard 5 used: [127.0.0.1:6527]
Shard 6 used: [127.0.0.1:6526]
Shard 7 used: [127.0.0.1:6523]
============== End of Results ==============
Cleanup
When the tests are done, stop the virtual devices.

From the root of the same tree where Cuttlefish was built, run the following commands:

source build/envsetup.sh

```
lunch aosp_cf_x86_64_only_phone-userdebug
stop_cvd
```

Let me explain this whole process in **simple words**, step by step:

🧪 Goal:

You want to **test your Android virtual device (Cuttlefish)** using **CTS** (Compatibility Test Suite) to ensure it's Android-compliant.

🔧 STEP 1: Build the Virtual Device (Cuttlefish)

This makes a virtual Android phone using your AOSP source code.

```bash
source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug
make -j
```

> ✅ **What this does:** Sets up the environment, selects Cuttlefish as your target, and builds it.

🧱 STEP 2: Build CTS

CTS is separate. Build it with:

```bash
source build/envsetup.sh
m -j cts WITH_DEXPREOPT_BOOT_IMG_AND_SYSTEM_SERVER_ONLY=false TARGET_PRODUCT=aosp_x86_64
```

> ✅ **What this does:** Builds the test suite (`cts`) for the same architecture.

🖥️ STEP 3: Launch 8 Virtual Devices (Multi-Tenant)

```bash
launch_cvd --num_instances=8 --daemon
```

> ✅ **This creates 8 Cuttlefish virtual devices** to run tests in parallel (faster).

Check they are running:

```bash
adb devices
```

You should see 8 devices like this:

```makefile
127.0.0.1:6520  device
127.0.0.1:6521  device
...
```

📶 STEP 4: Enable Wi-Fi on All Devices

CTS tests expect Wi-Fi to be ON, so run this:

```bash
for SERIAL in $(adb devices | grep -E '127.+device$' | cut -f1); do
  adb -s ${SERIAL} install -r -g tools/tradefederation/core/res/apks/wifiutil/WifiUtil.apk
  adb -s ${SERIAL} shell su root svc wifi enable
  adb -s ${SERIAL} shell am instrument -e method "connectToNetwork" -e scan_ssid "false" -e ssid
"VirtWifi" -w com.android.tradefed.utils.wifi/.WifiUtil
done
```

> ✅ **What this does:** Turns on Wi-Fi and connects it to a dummy network called **`VirtWifi`**.

🚀 STEP 5: Run CTS (Virtual Stable Subset)

```bash
cts-tradefed run cts-virtual-device-stable --no-enable-parameterized-modules --max-testcase-run-
count 2 --retry-strategy RETRY_ANY_FAILURE --reboot-at-last-retry --shard-count 8
```

> ✅ **What this does:**

- Runs a **special CTS test plan** (**`cts-virtual-device-stable`**) made for virtual devices
- Uses all 8 Cuttlefish instances in parallel (sharding)
- Retries failed tests

📌 Takes ~20–25 minutes and gives output like:

```yaml
Total Tests       : 92483
PASSED            : 92363
```

```
FAILED            : 0
IGNORED           : 71
```

🧹 STEP 6: Cleanup (Stop the Virtual Devices)

```bash
stop_cvd
```

🏢 Can OEMs and Vendors Use This?

✅ **Yes, definitely.**
This is especially useful for:

- **OEMs and SoC vendors** validating Android compliance before shipping
- **Developers** doing pre-integration testing on virtual devices
- Running **automated CTS on CI/CD servers** without needing physical devices

---

Let me know if you want a visual diagram or a shell script to automate all these steps.

how to vts here

To run **VTS (Vendor Test Suite)** on **Cuttlefish**, follow a process similar to CTS, but with key differences.

🧠 First, what is VTS?

VTS tests the **vendor** side (HALs, kernel, etc.), while **CTS** tests **framework + app compatibility**.
Cuttlefish helps run **VTS tests** in a virtual environment so you don't need physical hardware.

✅ Steps to Run VTS on Cuttlefish

---

1. Build Cuttlefish device image

```bash
source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug
```

```
make -j
```

2. Build VTS

From the same AOSP tree:

```bash
source build/envsetup.sh
m -j vts
```

3. Launch Cuttlefish instances

```bash
launch_cvd --num_instances=1 --daemon
```

You should see it via:

```bash
adb devices
```

4. Run VTS tests

Run VTS using **`vts-tradefed`**:

```bash
vts-tradefed
```

Inside the shell, run:

```bash
run vts --plan vts-core
```

Optional flags:

```bash
run vts --plan vts-core --shard-count 4 --retry-any-failure --reboot-at-last-retry
```

📘 Summary of Commands

```bash
# Set up build
source build/envsetup.sh
lunch aosp_cf_x86_64_only_phone-userdebug

# Build VTS and Cuttlefish
make -j vts

# Launch emulator
launch_cvd --daemon

# Run VTS
vts-tradefed
run vts --plan vts-core
```

🧹 Cleanup

After tests:

```bash
stop_cvd
```

Would you like the difference between CTS and VTS explained more simply?

> how vts can run on cf?

Great question! Let me explain in **very simple terms** how **VTS can run on Cuttlefish (CF)**:

💡 What is VTS?

VTS (Vendor Test Suite) tests things like:

- HALs (hardware interfaces)
- Kernel drivers
- Binder
- System partitions

It's designed to **test the "vendor" part of Android** — things below the framework level.

💡 What is Cuttlefish?

Cuttlefish is a **virtual Android device**.
It runs Android in a virtual machine on your PC/server.
So it's like a real device, but inside software.

✅ Why VTS can run on Cuttlefish

Because Cuttlefish is:

- A **real Android system image**
- With a **real vendor partition**
- That exposes **HALs and kernel**, just like a phone

So even though it's virtual, VTS can:

- Talk to the HALs
- Check kernel APIs
- Verify vendor interface

🛠️ How does it work?

You:

1. Build the **Cuttlefish image** (AOSP for Cuttlefish)
2. Launch the virtual device using `launch_cvd`
3. Build and run **VTS tests** using `vts-tradefed`
4. The VTS tests connect to the running Cuttlefish VM via **ADB**
5. Tests are sent to the virtual device and executed like a real phone

---

🔁 Flow (Visual)

```scss
[Your PC] ──> [Cuttlefish Android VM]
          │                    ▲
          └─ VTS tests ────────┘    (run via vts-tradefed)
```

⚠️ Note

- Cuttlefish must be running a **userdebug** build
- Some **hardware-specific VTS tests** (e.g., camera, GPS) may not work since CF is virtual
- But **Binder, HAL, kernel interface, SELinux, VNDK, VINTF** tests will work fine

---

Would you like a real example step-by-step?

Cuttlefish: Create a custom device

bookmark_border
This page describes how to create a customized Cuttlefish device. Cuttlefish includes predefined device types in different form factors as listed in AndroidProducts.mk. In addition to the general device customization options described in Adding a new device, you can make Cuttlefish-specific customizations such as virtual device board configuration presets (vsoc_x86_64, vsoc_arm64, vsoc_riscv64), kernel prebuilts, bootloader prebuilts, vendor properties, system configurations, nested virtualization support, and display options. For a full list of the build time parameters that can be customized, see
device/google/cuttlefish/vsoc_x86_64/phone/aosp_cf.mk.

The following steps describe how to create a fictional x86-64 big_phone device ten times the size of an ordinary Cuttlefish device.

Inherit from an existing target
To inherit from an existing target:

Create a device/google/cuttlefish/vsoc_x86_64/big_phone directory.
Create an aosp_cf.mk file in that directory.

$(call inherit-product, device/google/cuttlefish/vsoc_x86_64_phone.mk)

PRODUCT_NAME: big_phone
PRODUCT_DEVICE: vsoc_x86_64
PRODUCT_MANUFACTURER := My Company
PRODUCT_MODEL: My Company very large phone

PRODUCT_VENDOR_PROPERTIES += \
    ro.soc.manufacturer=$(PRODUCT_MANUFACTURER) \
    ro.soc.model=$(PRODUCT_DEVICE)
Add a lunch target
Insert the lunch target into the device/google/cuttlefish/AndroidProducts.mk file:

PRODUCT_MAKEFILES := \

...
big_phone:$(LOCAL_DIR)/vsoc_x86_64/big_phone/aosp_cf.mk
...

lunch big_phone
Define JSON configuration
To launch the Cuttlefish device, create a JSON configuration file named big_phone.json with a hierarchical structure representing the device properties. For example, in the JSON configuration file, you can specify options such as the RAM allocated for the VM and the display configuration. This file doesn't have to be in the AOSP tree. For details on the JSON format for configurations, see Canonical configurations.

```
{
  "instances":
     [
       {
         "vm": {
           "memory_mb": 40960,
         },
         "graphics": {
           "displays": [
             {
               "width": 7200,
               "height": 12800,
               "dpi": 320
             }
           ]
         }
       }
     ]
}
```

To launch the configuration, run:

```
cvd create --config_file=big_phone.json
```

Run launch_cvd (legacy)
Certain configuration properties aren't available in the hierarchical JSON configuration format. For such configurations, you can set launch_cvd flag default values from a single-level JSON dictionary. For a full list of all configuration options, see cf_flags_validator.cpp.

The following describes an example of how to override the default values of the launch_cvd flag options using a JSON configuration file and enable the Cuttlefish launcher to launch the custom configuration.

Create a JSON configuration file, device/google/cuttlefish/shared/config/config_big_phone.json, with custom values.

```
{
  "x_res": 7200,
  "y_res": 12800,
  "dpi": 320,
  "memory_mb": 40960,
  "ddr_mem_mb": 49150,
}
```

For the Cuttlefish launcher to launch the big_phone config, ensure it has access to the device/google/cuttlefish/shared/config/config_big_phone.json file by doing the following:

Declare the JSON artifact as a build artifact by adding the prebuilt_etc_host stanza in the device/google/cuttlefish/shared/config/Android.bp file.

```
prebuilt_etc_host {
    name: "cvd_config_big_phone.json",
    src: "config_big_phone.json",
    sub_dir: "cvd_config",
}
```

Add the resulting build artifact declaration into the Cuttlefish launcher by running the following in device/google/cuttlefish/shared/device.mk.

```
$(call soong_config_append,cvd,launch_configs,cvd_config_big_phone)
```

Create an android_info.txt file and tie the big_phone config to the file by adding the following line to device/google/cuttlefish/vsoc_x86_64/big_phone/aosp_cf.mk:

```
TARGET_BOARD_INFO_FILE :=
device/google/cuttlefish/vsoc_x86_64/<var>big_phone</var>/android-info.txt
```

Label the device type with the big_phone config by populating device/google/cuttlefish/vsoc_x86_64/big_phone/android-info.txt with the following:

```
config=big_phone
```

Let’s break it down **in very simple words** to understand **how to create and run a custom Cuttlefish (CF) device**, and **how it relates to VTS (Vendor Test Suite)**.

🧠 What is Cuttlefish?

Cuttlefish is a **virtual Android device** that runs on your PC. It's used for testing Android without needing a real phone.

🛠️ What is this “Create a custom Cuttlefish device” guide doing?

Imagine you want to test a very big Android phone (say, 7200x12800 display!) on your computer. This guide helps you do that by:

🔧 1. Creating a new device configuration

You make a folder and tell it:

- Use `x86_64` as CPU type
- Use big screen and more RAM

This is done by:

- Copying an existing config (`vsoc_x86_64_phone.mk`)
- Creating a new `.mk` file (`aosp_cf.mk`)
- Setting values like `PRODUCT_MODEL := very large phone`

🍱 2. Adding a new "lunch" target

You edit `AndroidProducts.mk` to add your new device config:

```make
big_phone:$(LOCAL_DIR)/vsoc_x86_64/big_phone/aosp_cf.mk
```

Then you can build your device using:

```bash
lunch big_phone
```

📦 3. Create a config file (JSON)

This JSON file tells Cuttlefish:

- How much memory the virtual phone should use
- What the screen size should be

```json
{
  "instances": [
    {
      "vm": {
        "memory_mb": 40960
      },
      "graphics": {
        "displays": [
          {
            "width": 7200,
            "height": 12800,
            "dpi": 320
          }
        ]
      }
    }
  ]
}
```

To start the device:

```bash
cvd create --config_file=big_phone.json
```

✅ How does VTS run on this?

Once your **custom CF device is running**, you can run **VTS tests on it**, just like on a real device.

Steps:

1. **Launch the CF device**:

   ```bash
   launch_cvd
   ```

2. **Make sure it's visible to `adb`**:

   ```bash
   adb devices
   ```

3. **Run VTS** using `vts-tradefed`:

   ```bash
   vts-tradefed
   ```

And inside the shell:

```bash
run vts -m VtsHalCameraProviderV2_4Target
```

You can run **any HAL or system test** this way — because the CF behaves like a real Android phone.

💡 Summary

Step	What You Do
Create **`aosp_cf.mk`**	Defines your new device
Add to **`AndroidProducts.mk`**	So you can **`lunch big_phone`**
Write JSON	Defines how much RAM/screen size
`cvd create` + **`launch_cvd`**	Launch the virtual phone
`adb` + **`vts-tradefed`**	Run VTS on it

Would you like me to show you how to write a minimal working VTS command on this?

Develop Android kernels

bookmark_border
This page describes how to develop Android kernels on Cuttlefish devices.

Supported kernel manifests
Cuttlefish supports the following kernel manifests on the Android latest release branch and AOSP GSI releases.

On android-latest-release
common-android-mainline
On android-15.0.0_r36
common-android16-6.12
common-android15-6.6
common-android14-6.1
common-android14-5.15
On aosp-android13-gsi
common-android13-5.15
common-android13-5.10
On aosp-android12-gsi
common-android12-5.10

common-android12-5.4
common-android-4.19-stable
On aosp-android11-gsi
common-android11-5.4
common-android-4.19-stable
common-android-4.14-stable
The following are kernel sources for kernel development on Cuttlefish devices.

kernel/common-android-mainline
kernel/common-android16-6.12
kernel/common-android15-6.6
kernel/common-android14-6.1
kernel/common-android14-5.15
kernel/common-android13-5.15
kernel/common-android13-5.10
Develop Android kernels
To develop kernels on Cuttlefish devices, follow these steps:

Clone the appropriate kernel repo manifest for your Cuttlefish device and make any changes.

```
mkdir common-android-mainline
cd common-android-mainline
repo init -u https://android.googlesource.com/kernel/manifest -b common-android-mainline
repo sync -c -j$(nproc) -q
```

Build the kernel and ramdisk. For more information, see Building the Vendor Modules for the Virtual Device.

```
tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist
```

If using common-android-4.19-stable, common-android-4.14-stable, or common-android11-5.4 kernel manifests, run:

```
BUILD_CONFIG=common/build.config.gki.x86_64 build/build.sh && BUILD_CONFIG=common-modules/virtual-device/build.config.cuttlefish.x86_64 build/build.sh
```

Instead of building locally (to avoid downloading the kernel source, setting up a kernel development environment), you can download the relevant artifacts at ci.android.com.

android-mainline
android16-6.12
android15-6.6
android14-6.1
android14-5.15
android13-5.15
android13-5.10

android12-5.10
android12-5.4
android11-5.4
Launch the kernel and ramdisk with Cuttlefish.

cvd create \
  -kernel_path=./out/virtual_device_x86_64/dist/bzImage \
  -initramfs_path=./out/virtual_device_x86_64/dist/initramfs.img
If launching a non-GKI kernel (anything at or under 4.19), omit the -initramfs_path argument.

If you're launching the kernel on an aarch64 version of Cuttlefish, use the aarch64 build configuration and use the Image artifact instead of bzImage.

cvd create \
  -kernel_path=./out/virtual_device_aarch64/dist/Image \
  -initramfs_path=./out/virtual_device_aarch64/dist/initramfs.img

Here's a **simple explanation** of how to **run VTS (Vendor Test Suite)** on a **Cuttlefish (CF)** virtual device:

✅ What is Cuttlefish?

Cuttlefish is a **virtual Android device** that runs on your PC (Linux). It behaves like a real Android device, and it's mainly used for **testing** Android platform features.

✅ What is VTS?

VTS is a test suite used to verify that **vendor implementations** (e.g., HALs, drivers, kernel configs) are compatible with Android standards.

🧩 How VTS Works with Cuttlefish

1. Set up Cuttlefish

Make sure you have a **Cuttlefish device running** on your machine.

```bash
launch_cvd --start_webrtc=true
```

If you want to develop/test a custom kernel or HAL, launch it using:

```bash
cvd create \
  -kernel_path=./out/virtual_device_x86_64/dist/bzImage \
  -initramfs_path=./out/virtual_device_x86_64/dist/initramfs.img
```

Then run:

```bash
cvd start
```

---

2. Start WebRTC (optional)

To interact with your Cuttlefish in the browser:

Visit https://localhost:8443 in your browser.

---

3. Start VTS Host

On your **host machine** (Linux), run VTS against the running Cuttlefish:

```bash
vts-tradefed
```

This starts the **VTS test runner**.

---

4. Run Tests

You can list and run specific tests, for example:

```bash
run vts --module VtsKernelConfigTest
```

Or for a more complete test set:

```bash
run vts
```

---

5. ADB Works Normally

Since Cuttlefish supports **`adb`**, you can use **`adb shell`**, **`adb install`**, etc., just like on a real device.

🔧 Use Cases

Use Case	Can OEM/Vendor Use This?
Kernel testing	✅ Yes
HAL testing (via VTS)	✅ Yes
Platform debugging	✅ Yes
GSI validation	✅ Yes

🔐 Ports to Open (if remote)

If using WebRTC or VTS remotely, make sure to open these firewall ports:

- TCP: **`8443`**, **`15550–15599`**
- UDP: **`15550–15599`**

Would you like a full command sequence to:

- Build kernel
- Launch Cuttlefish
- Run VTS?

Let me know your use case (kernel testing, HAL testing, or GSI testing), and I'll walk you through it step-by-step.

Cuttlefish-1

Develop Android kernels

bookmark_border
This page describes how to develop Android kernels on Cuttlefish devices.

Supported kernel manifests
Cuttlefish supports the following kernel manifests on the Android latest release branch and AOSP GSI releases.

On android-latest-release
common-android-mainline
On android-15.0.0_r36
common-android16-6.12
common-android15-6.6
common-android14-6.1
common-android14-5.15
On aosp-android13-gsi
common-android13-5.15
common-android13-5.10
On aosp-android12-gsi
common-android12-5.10
common-android12-5.4
common-android-4.19-stable
On aosp-android11-gsi
common-android11-5.4
common-android-4.19-stable
common-android-4.14-stable
The following are kernel sources for kernel development on Cuttlefish devices.

kernel/common-android-mainline
kernel/common-android16-6.12
kernel/common-android15-6.6
kernel/common-android14-6.1
kernel/common-android14-5.15
kernel/common-android13-5.15
kernel/common-android13-5.10
Develop Android kernels
To develop kernels on Cuttlefish devices, follow these steps:

Clone the appropriate kernel repo manifest for your Cuttlefish device and make any changes.

mkdir common-android-mainline
cd common-android-mainline
repo init -u https://android.googlesource.com/kernel/manifest -b common-android-mainline
repo sync -c -j$(nproc) -q
Build the kernel and ramdisk. For more information, see Building the Vendor Modules for the Virtual Device.

tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist
If using common-android-4.19-stable, common-android-4.14-stable, or common-android11-5.4 kernel manifests, run:

BUILD_CONFIG=common/build.config.gki.x86_64 build/build.sh && BUILD_CONFIG=common-modules/virtual-device/build.config.cuttlefish.x86_64 build/build.sh
Instead of building locally (to avoid downloading the kernel source, setting up a kernel development environment), you can download the relevant artifacts at ci.android.com.

android-mainline
android16-6.12
android15-6.6
android14-6.1
android14-5.15
android13-5.15
android13-5.10
android12-5.10
android12-5.4
android11-5.4
Launch the kernel and ramdisk with Cuttlefish.

cvd create \
    -kernel_path=./out/virtual_device_x86_64/dist/bzImage \
    -initramfs_path=./out/virtual_device_x86_64/dist/initramfs.img
If launching a non-GKI kernel (anything at or under 4.19), omit the -initramfs_path argument.

If you're launching the kernel on an aarch64 version of Cuttlefish, use the aarch64 build configuration and use the Image artifact instead of bzImage.

cvd create \
    -kernel_path=./out/virtual_device_aarch64/dist/Image \
    -initramfs_path=./out/virtual_device_aarch64/dist/initramfs.img

Here's a **step-by-step explanation** for how to **develop Android kernels on Cuttlefish virtual devices**, based on the official guide you referenced.

✅ 1. Choose the Right Kernel Manifest

Cuttlefish supports many kernel versions depending on the Android release you're targeting:

Android Release	Supported Kernels
android-latest-release	`common-android-mainline`
android-15	`common-android16-6.12`, `common-android15-6.6`
android-14	`common-android14-6.1`, `common-android14-5.15`
android-13	`common-android13-5.15`, `common-android13-5.10`
android-12	`common-android12-5.10`, `common-android12-5.4`
android-11	`common-android11-5.4`, `common-android-4.19-stable`, `common-android-4.14-stable`

✅ 2. Clone the Kernel Source

Choose a kernel repo based on the branch you want (e.g. `common-android-mainline`):

```bash
mkdir common-android-mainline
cd common-android-mainline
repo init -u https://android.googlesource.com/kernel/manifest -b common-android-mainline
repo sync -c -j$(nproc) -q
```

- `repo init`: Initializes your local repo with Google's kernel manifest.
- `repo sync`: Downloads kernel source code.

✅ 3. Build the Kernel + Ramdisk

✅ If you're using a GKI kernel (Generic Kernel Image):

```bash
tools/bazel run //common-modules/virtual-device:virtual_device_x86_64_dist
```

- This builds:

- **`bzImage`**: kernel image
- **`initramfs.img`**: initramfs

✅ If you're using an older kernel (non-GKI) like `4.19`, `4.14`, etc.:

```bash
BUILD_CONFIG=common/build.config.gki.x86_64 build/build.sh
BUILD_CONFIG=common-modules/virtual-device/build.config.cuttlefish.x86_64 build/build.sh
```

These **`build.config`** files tell the build system how to configure and compile the kernel and modules.

✅ Alternative: Download Prebuilt Kernels

If you want to avoid building locally, download prebuilt kernels from:
👉 https://ci.android.com/builds/branches/

Look for:

- **`android-mainline`**
- **`android16-6.12`**, etc.

✅ 4. Launch Cuttlefish with Your Kernel

🖥️ For x86_64 builds:

```bash
cvd create \
  -kernel_path=./out/virtual_device_x86_64/dist/bzImage \
  -initramfs_path=./out/virtual_device_x86_64/dist/initramfs.img
```

📱 For ARM64 (aarch64) builds:

```bash
cvd create \
  -kernel_path=./out/virtual_device_aarch64/dist/Image \
  -initramfs_path=./out/virtual_device_aarch64/dist/initramfs.img
```

> ⚠️ **NOTE**: If you're using a non-GKI kernel like `4.19`, **omit** the **`-initramfs_path`** argument.

🛠️ Summary

Step	Action
1	Choose your kernel version (**`common-android14-6.1`**, etc.)
2	**`repo init`** + **`repo sync`** to download source
3	Build kernel using **`bazel`** (for GKI) or **`build.sh`** (for old versions)
4	Launch kernel with **`cvd create`**

💡 Tips

- **Use `bazel`** only for kernels that support it (most GKI branches).
- To **customize the kernel**, make changes in the source tree before building.
- Use **`--console=true`** or **`cvd start`** to boot and interact with the virtual device after creation.

Would you like a **sample patch**, help with **adding a kernel module**, or instructions on **debugging the boot process** in Cuttlefish?

Develop Android bootloader features

bookmark_border
This page describes how to build, run, and develop Android bootloader features on a Cuttlefish device.

AOSP sources
To develop Android bootloader features, use the Cuttlefish configuration of U-boot with Cuttlefish in the latest release branch (android16-release) or on generic system image (GSI) branches for Android 11 or higher. The following are the sources in AOSP:

Manifest for the Cuttlefish configuration of U-boot (u-boot-mainline).

U-boot bootloader source.

x86 Cuttlefish default bootloader.

aarch64 Cuttlefish default bootloader.

QEMU bootloaders (arm, aarch64, x86_64, riscv)

Develop a bootloader
To build, run, and develop bootloader features locally, follow these steps:

Note: Instead of building locally (to avoid downloading the U-boot source and setting up a bootloader development), you can download the relevant artifacts at ci.android.com.
Clone the manifest for the Android fork of U-boot:

```
$ mkdir u-boot-mainline
$ cd u-boot-mainline
$ repo init -u https://android.googlesource.com/kernel/manifest -b u-boot-mainline
$ repo sync -j$(nproc) -q
```

Build the bootloader target appropriate for your Cuttlefish architecture. For example:

x86_64

```
$ tools/bazel run //u-boot:crosvm_x86_64_dist
```

aarch64

```
$ tools/bazel run //u-boot:crosvm_aarch64_dist
```

The resulting bootloader binary (u-boot.bin for arm64 and u-boot.rom for x86_64) is found in $PATH_TO_REPO/out/u-boot-mainline/dist.

Launch the Cuttlefish device with the bootloader. Include the pause_in_bootloader parameter to disable autoboot and let you interact with the bootloader.

```
$ launch_cvd \
    -bootloader /$PATH/$TO/u-boot-mainline/out/u-boot-mainline/dist/u-boot.rom \
    -pause_in_bootloader -console=true
```

Note: If on aarch64, use u-boot.bin instead of u-boot.rom.
To interact with the bootloader, connect to the bootloader console.

```
$ screen ~/cuttlefish_runtime/console
```

When done with bootloader interactions, continue the boot by typing boot into the console.

Test bootloader flow with different kernels
To test the bootloader flow with different kernels or kernel modules, pass in the target artifacts (kernel and kernel modules) using the --kernel_path and --initramfs_path arguments. The assembler repacks the boot images based on the artifacts passed in the arguments.

```
cvd create \
    -kernel_path=/$PATH/$TO/common-android14-6.1/out/android14-6.1/dist/bzImage \
    -initramfs_path=/$PATH/$TO/common-android14-6.1/out/android14-6.1/dist/initramfs.img
```

For more information, see Develop Android Kernels.

Here is a clear breakdown of how to **develop Android bootloader features on Cuttlefish** using the U-Boot source from AOSP.

🔧 What You're Doing

You're modifying or testing the **U-Boot bootloader** for Cuttlefish — a virtual Android device that runs on x86_64 or ARM64 (aarch64). U-Boot is responsible for loading the kernel and booting the OS.

📁 AOSP Bootloader Sources

You'll work with the following from AOSP:

Source	Description
`u-boot-mainline` manifest	Manifest to sync U-Boot sources
`u-boot` source tree	The actual U-Boot code
`x86_64` & **`aarch64`** default bootloaders	Default platform configs for Cuttlefish
QEMU bootloaders	For simulating bootloaders (optional)

🧑‍💻 Step-by-Step: Develop Android Bootloader Features

✅ 1. Clone U-Boot Source

```bash
mkdir u-boot-mainline
cd u-boot-mainline
repo init -u https://android.googlesource.com/kernel/manifest -b u-boot-mainline
repo sync -j$(nproc) -q
```

This fetches the **U-Boot manifest and source code**.

✅ 2. Build U-Boot for Cuttlefish

Depending on your architecture:

For x86_64:

```bash
tools/bazel run //u-boot:crosvm_x86_64_dist
```

For aarch64:

```bash
tools/bazel run //u-boot:crosvm_aarch64_dist
```

Output is placed in:

```bash
out/u-boot-mainline/dist/u-boot.rom    # for x86_64
out/u-boot-mainline/dist/u-boot.bin    # for aarch64
```

✅ 3. Run Cuttlefish with Custom Bootloader

Launch Cuttlefish and **pause in the bootloader** to interact with it:

```bash
launch_cvd \
  -bootloader ./out/u-boot-mainline/dist/u-boot.rom \
  -pause_in_bootloader \
  -console=true
```

For ARM64:

```bash
launch_cvd \
  -bootloader ./out/u-boot-mainline/dist/u-boot.bin \
  -pause_in_bootloader \
  -console=true
```

✅ 4. Connect to the Bootloader Console

Use **`screen`** or **`minicom`** to interact with the U-Boot prompt:

```bash
screen ~/cuttlefish_runtime/console
```

Type commands like:

```bash
=> printenv
=> help
=> boot
```

When done testing, continue booting Android with:

```bash
=> boot
```

✅ 5. Test Bootloader with Different Kernel

You can test booting **different kernel images**:

```bash
cvd create \
  -kernel_path=./common-android14-6.1/out/android14-6.1/dist/bzImage \
  -initramfs_path=./common-android14-6.1/out/android14-6.1/dist/initramfs.img
```

This lets you verify how U-Boot interacts with different kernel configurations.

🧠 Notes

- You **don't need to build Android** to test U-Boot; just the bootloader + kernel/initramfs.
- U-Boot scripts or config variables like **`bootcmd`**, **`bootargs`**, and **`env`** can be used to modify boot flow.
- You can **add custom bootloader features**, like secure boot stubs, command hooks, logging, etc., in U-Boot C files.

💡 Optional: Download Prebuilt Bootloader

Skip local builds using:
👉 https://ci.android.com/builds/branches/ — choose **`u-boot-mainline`**.

🧪 Summary

Step	Action
1	Clone U-Boot: **`repo init -b u-boot-mainline`**
2	Build for x86_64 or aarch64 using Bazel

Step	Action
3	Launch Cuttlefish with **`-bootloader`** and **`-pause_in_bootloader`**
4	Connect via **`screen`** to interact with U-Boot
5	Boot a custom kernel using **`cvd create`** arguments

Would you like help with:

- Adding a **custom U-Boot command**?
- Debugging **boot flow issues**?
- Modifying the **bootargs** or environment?

Let me know and I'll guide you through that too.

Cuttlefish: Display hotplug

bookmark_border
Cuttlefish virtual devices can handle display hotplugs, which allow devices to connect and disconnect displays at runtime, by using the cvd_display command-line interface.

Warning: Input isn't supported on hotplugged displays.
Add
To add displays, run:

cvd display add \
--display=width=WIDTH-0,height=HEIGHT-0 \
--display=width=WIDTH-1,height=HEIGHT-1
List
To list the current displays and their display IDs, run:

cvd display list
Remove
To remove displays, run:

cvd display remove \
--display=FIRST-DISPLAY-ID \
--display=SECOND-DISPLAY-ID \
...

The display IDs can be found using the cvd display list command.

Here's a **simple guide** to using **Cuttlefish's display hotplug** feature to **add, list, and remove virtual displays at runtime** using the **`cvd display`** command-line tool.

🖥️ What is Display Hotplug in Cuttlefish?

Display hotplug allows you to **dynamically add or remove displays** (monitors/screens) while the Cuttlefish virtual device is running — **without rebooting** the device.

> ⚠️ **Note:** Input (touch, mouse, etc.) is **not supported** on these hotplugged displays — they are output-only.

✅ Add Displays

You can add one or more displays by specifying their resolution:

```bash
cvd display add \
  --display=width=1280,height=720 \
  --display=width=1920,height=1080
```

This adds:

- A 1280x720 display
- A 1920x1080 display

Each gets assigned a **unique Display ID**.

You can also specify DPI, density, or other parameters if supported in future builds.

🔍 List Active Displays

To view currently attached displays and their **display IDs**:

```bash
cvd display list
```

Output example:

```yaml
Display ID: 0  (built-in)
Display ID: 1  (1280x720)
Display ID: 2  (1920x1080)
```

Use these IDs to manage displays (e.g., remove).

❌ Remove Displays

To remove a specific display using its ID:

```bash
cvd display remove \
  --display=1 \
  --display=2
```

This removes Display ID 1 and 2, keeping the primary (ID 0) untouched.

> 💡 You cannot remove the default built-in display (ID 0).

🧪 Example Full Flow

```bash
# Add two new displays
cvd display add \
  --display=width=1024,height=768 \
  --display=width=1920,height=1080

# List all displays
cvd display list

# Remove added displays (e.g., ID 1 and 2)
cvd display remove \
  --display=1 \
  --display=2
```

📦 When to Use This

- Test multi-display Android behavior
- Validate app behavior on different screen sizes
- Simulate monitor docking scenarios

Would you like help **automating this in a script**, or testing **multi-display rendering** inside Android?

Cuttlefish: Multi-tenancy

bookmark_border
Cuttlefish multi-tenancy allows for your host machine to launch multiple virtual guest devices with a single launch invocation.

These multiple Cuttlefish virtual machines can share some of the host disk resources, allowing you to conserve disk space. Each virtual machine gets a distinct disk overlay covering the Android partitions (such as super, userdata, boot). Any modification to the guest disk is placed in a guest-specific disk overlay, giving each Cuttlefish virtual machine a copy-on-write view of its disk partitions. As each virtual device writes more data to its view of the disk, the overlay files on the host disk increase in size because they're capturing the delta between the original state and their current state.

The overlay files can be reset to return the Cuttlefish disk to its original state. This process is known as powerwashing.

Instance numbers
Cuttlefish devices create and consume resources on the host tied to their instance number. For example, an adb connection is exposed through a TCP server socket at port 6520 for the Cuttlefish device with instance number 1. When multiple instances are launched, it creates a TCP server socket at port 6520 for the first device, 6521 for the second device, and increments the port number for each additional device.

Modify maximum number of supported instances
Host-side resources, such as TAP network devices, must be preallocated for Cuttlefish virtual machines. By default, the cuttlefish-base Debian package provisions resources for instance numbers 1 through 10.

This can be changed through modifying num_cvd_accounts in /etc/default/cuttlefish-host-resources and restarting the cuttlefish-host-resources service by running the following command:

sudo systemctl restart cuttlefish-host-resources
Canonical configurations
Canonical configurations represent the configuration for Cuttlefish devices in JSON format. You can create canonical configuration JSON files to describe the configuration of multiple Cuttlefish devices in a multi-tenant scenario.

The following describes the format used for canonical configuration JSON files (replace placeholders with device configurations).

```
{
  "common": {
    CONFIGURATIONS_APPLYING_TO_ALL_DEVICES
  },
  "instances": [
    {
      FIRST_DEVICE_CONFIGURATION
    },
    {
      NTH_DEVICE_CONFIGURATION
    }
  ]
}
```

The following is an example configuration file of a setup with two devices, a phone device and a wearable device:

```
{
  "instances": [
    {
      "@import": "phone",
      "vm": {
        "memory_mb": 8192,
        "setupwizard_mode": "OPTIONAL",
        "cpus": 4
      },
      "disk": {
        "default_build": "/home/username/devices/cf_x86_64_only_phone-userdebug"
      }
    },
    {
      "@import": "wearable",
      "vm": {
        "memory_mb": 8192,
        "setupwizard_mode": "REQUIRED",
        "cpus": 4
      },
      "disk": {
        "default_build": "/home/username/devices/cf_gwear_x86-userdebug"
      }
    }
  ]
}
```

This table describes the elements used in the example configuration with two devices:

Element Type Description
instances Array Array of JSON objects, each describing a single device in the multi-tenant group of devices to be launched.
@import String Indicates the device type and sets default values for the device configuration. The supported device types are phone and wearable.
vm Object Specifies properties that apply to the VM environment specific to the device instance.
memory_mb Number The amount of memory to allocate for a specific VM in megabytes.
setupwizard_mode String Specifies whether the device setup wizard is run for the user at first run of the device. The supported options for setup wizard mode are DISABLED, OPTIONAL, and REQUIRED.
cpus Number The number of virtual CPUs allocated to the VM.
disk Object Specifies properties relevant to locating file system artifacts used for running a single VM instance.
default_build String Local file system path to a folder containing a Cuttlefish virtual device image.
For more examples of configuration files distributed with cvd, see the cvd_test_configs folder in the Cuttlefish code tree.

Control multi-tenant instances
This section describes how to create and destroy devices in a multi-tenant scenario. You can launch multiple Cuttlefish devices using the following options:

Canonical config: Launch customized configurations for each device using a JSON config file describing the list of instances to launch.
Single guest image: Launch multiple devices from a single guest image.
Start multiple VMs from canonical config
To create multiple VMs using a canonical configuration that describes the multi-tenant scenario, invoke the cvd create command with the --config_file= flag prefix as follows:

cvd create --config_file=CONFIG_FILE
The following is an example invocation referencing a canonical config file named /etc/phone.json.

cvd create --config_file=/etc/phone.json
Start multiple VMs from single guest image
To launch multiple Cuttlefish devices using a single guest image, use the --num_instances=N flag, where N is the number of devices to launch. By default, devices start from the instance number 1.

cvd create --num_instances=N
To change the starting instance number, do one of the following:

Add the --base_instance_num=N flag with N as the first instance number.

cvd create --base_instance_num=N
Launch Cuttlefish with a user account matching vsoc-NN, where NN is the base instance number in two digit form. For example, to launch Cuttlefish with a user account vsoc-01, run:

cvd create --base_instance_num=1
How to stop devices
Note: When used in a multi-tenant environment, the stop procedure is applied to all devices simultaneously.
To stop all the devices launched by the last cvd create invocation, run:

cvd stop
To restart devices that have been stopped, run:

cvd start
To completely remove a device from the system run:

cvd remove

Here's a complete and easy-to-follow breakdown of **Cuttlefish Multi-Tenancy**—how to **launch, manage, and configure multiple Android virtual devices (VMs)** on a single host.

🌐 What Is Multi-Tenancy in Cuttlefish?

Multi-tenancy allows you to **run multiple Android VMs (guests)** on one physical machine (host) **with one command**, where:

- All VMs **share base disk resources** (to save space).
- Each VM has a **private overlay** (copy-on-write) for its changes.
- All VMs are individually addressable (ADB, console, etc.).
- They can simulate **different device types** (e.g., phone + wearable).

📦 How Disk Overlays Work

Each VM uses:

- **Shared base image** (e.g., system.img, vendor.img)
- **Unique overlay image** (stores only changes per instance)

This **saves space** and allows **resetting** VMs without touching the base image — called **powerwashing**.

🔢 Instance Numbers

Every Cuttlefish VM is assigned a unique **instance number**.

Instance #	ADB Port
1	6520
2	6521
...	...

⚙️ Increase Number of Allowed Instances

By default, Cuttlefish supports up to 10 instances. To raise the limit:

1. Edit:

```bash
sudo nano /etc/default/cuttlefish-host-resources
```

2. Set:

```ini
num_cvd_accounts=20
```

3. Restart the service:

```bash
sudo systemctl restart cuttlefish-host-resources
```

🧾 Canonical Configuration JSON

Instead of repeating **`cvd create`** for every device, use a **canonical config file**.

📄 Example: `multi_device.json`

```json
{
  "instances": [
    {
      "@import": "phone",
      "vm": {
        "memory_mb": 8192,
        "cpus": 4,
        "setupwizard_mode": "OPTIONAL"
      },
      "disk": {
        "default_build": "/home/username/devices/cf_x86_64_only_phone-userdebug"
      }
    },
    {
      "@import": "wearable",
      "vm": {
        "memory_mb": 8192,
        "cpus": 4,
        "setupwizard_mode": "REQUIRED"
      },
      "disk": {
        "default_build": "/home/username/devices/cf_gwear_x86-userdebug"
      }
    }
  ]
}
```

Launch with Canonical Config:

```bash
cvd create --config_file=/path/to/multi_device.json
```

🧑‍💻 Launch Multiple VMs (Alternative: Single Image)

If you don’t want a config file and want to launch N identical VMs:

```bash
cvd create --num_instances=4
```

Change starting instance number:

```bash
cvd create --num_instances=4 --base_instance_num=5
```

User account **`vsoc-05`** will be created for instance 5, etc.

🛑 Manage Multi-Tenant VMs

Action	Command
Stop all VMs	`cvd stop`
Restart VMs	`cvd start`
Delete VMs	`cvd remove`
List running VMs	`ps -ef
ADB connect	`adb connect 127.0.0.1:6520`

✅ Summary Table

Feature	How
Add multiple devices	`cvd create --num_instances=N` or use JSON
Customize config	Use canonical config JSON file
Save space	Shared base image, copy-on-write overlays
Reset changes	Powerwashing by deleting overlay files
Port mapping	ADB: 6520 + instance_num - 1
Device types	`@import: phone`, `wearable`
Stop/remove devices	`cvd stop`, `cvd remove`

Would you like help **generating a config file dynamically**, **launching different Android builds per instance**, or **testing communication between VMs**?

Cuttlefish: Environment control

bookmark_border
This page describes how to control the environment in a Cuttlefish device using the REST API or the command line interface. For example, you can modify the Wi-Fi signal or update the GPS location of the Cuttlefish device.

Services
The cvd env command provides the following services for controlling the Cuttlefish environment:

Services Description
GnssGrpcProxy Controls GNSS for the Cuttlefish geolocation feature.
OpenwrtControlService Controls Openwrt, a virtualized Wi-Fi AP provider for the Cuttlefish Wi-Fi feature.

WmediumdService Controls Wmediumd, a wireless medium simulator for the Cuttlefish Wi-Fi feature.
CasimirControlService Controls Casimir, a simulator for external NFC devices.
Control the environment using the REST API
This section describes how to control the environment using the Rest API through the <https://localhost:1443> service endpoint.

List available services or methods
To get a list of all services, send a GET request to the following URL with a device ID.

https://localhost:1443/devices/DEVICE_ID/services
To get a list of all the methods for a service, send a GET request to the following URL with a device ID and the service name.

https://localhost:1443/devices/DEVICE_ID/services/SERVICE_NAME
To get detailed information for a method, such as the request or response message types, send a GET request to the following URL with a device ID, a service name, and the method name.

https://localhost:1443/devices/DEVICE_ID/services/SERVICE_NAME/METHOD_NAME
Get detailed information on request and response types
To get detailed information of a request or response message type, send a GET request to the following URL with a device ID, a service name, and the request or response message type. This prints all the names and types of each field in the message. You can then use this information to write a JSON-formatted proto message for sending a RPC request.

https://localhost:1443/devices/DEVICE_ID/services/SERVICE_NAME/REQUEST_OR_RESPONSE_TYPE_NAME/type
Send RPC request to make changes to the environment
To send a RPC request to call a method of a service with a JSON-formatted proto, send a POST request to the following URL with a device ID, a service name, and the method name. The JSON-formatted proto must be included in the body.

https://localhost:1443/devices/DEVICE_ID/services/SERVICE_NAME/METHOD_NAME
Example use case
The following is an example use case of the Rest API for modifying the Wi-Fi signal strength by calling SetTxpower.

Determine the service name, method name, and the request message type name for modifying the Wi-Fi signal strength.

Get a list of all available services by sending a GET request to the following URL.

https://localhost:1443/devices/cvd-1/services
This is an example response.

{"services":["OpenwrtControlService","EchoService","GnssGrpcProxy","WmediumdService"]}
Get a list of methods for WmediumdService by sending a GET request to the following URL.

https://localhost:1443/devices/cvd-1/services/WmediumdService
This is an example response.

{"methods":
["ListStations","LoadConfig","ReloadConfig","SetCivicloc","SetLci","SetPosition","SetSnr","SetTxpower","StartPcap","StopPcap"]}
Get information on the request and response message types for the SetTxpower method by sending a GET request to the following URL.

https://localhost:1443/devices/cvd-1/services/WmediumdService/SetTxpower
This is an example response.

{"request_type_name":"wmediumdserver.SetTxpowerRequest","response_type_name":"google.protobuf.Empty"}
Get detailed information for the wmediumdserver.SetTxpowerRequest request message type by sending a GET request to the following URL.

https://localhost:1443/devices/cvd-1/services/WmediumdService/wmediumdserver.SetTxpowerRequest/type
This is an example response.

```
message SetTxpowerRequest {
  string mac_address = 1;
  int32 tx_power = 2;
}
```
Send a RPC request to the WmediumdService service to modify the Wi-Fi signal strength to the desired level by sending a POST request to this URL with the following request body.

https://localhost:1443/devices/cvd-1/services/WmediumdService/SetTxpower

{"mac_address":"42:00:00:00:00:00", "tx_power":1}
This is an example response.

{}
Control the environment using the command line
This section describes the subcommands available for the cvd env CLI command. For further details, print the help message using cvd help env.

List available services or methods
To get a list of all services, use cvd env ls without any arguments.

cvd env ls
To get a list of all the methods for a service, include the name of the service as an argument.

cvd env ls SERVICE_NAME
To get detailed information such as the request or response message types of a method, include the service name and method name.

cvd env ls SERVICE_NAME METHOD_NAME
Get detailed information on request and response types
To get detailed information of a request or response message type, use the cvd env type command. This command prints all names and types of each field in the message. You can then use this information to write a JSON-formatted proto message for sending a RPC request.

cvd env type SERVICE_NAME REQUEST_OR_RESPONSE_TYPE_NAME
Send RPC request to make changes to the environment
To send a RPC request to call a method of a service with a JSON-formatted proto, use the cvd enc call command. When the RPC request terminates, the interface prints the message Rpc succeeded with OK status and, if available, a response message including values.

cvd env call SERVICE_NAME METHOD_NAME JSON_FORMATTED_PROTO
Example use case
The following is an example use case of the cvd env CLI command for modifying the Wi-Fi signal strength by calling SetTxpower.

Determine the service name, method name, and the request message type name for modifying the Wi-Fi signal strength.

Get a list of all available services.

```
cvd env ls
(Omitted)
{
  "services" :
  [
    "OpenwrtControlService",
    "EchoService",
    "GnssGrpcProxy",
    "WmediumdService"
  ]
}
```

Get a list of methods for WmediumdService.

```
cvd env ls WmediumdService
(Omitted)
{
  "methods" :
  [
    "ListStations",
    "LoadConfig",
    "ReloadConfig",
    "SetCivicloc",
    "SetLci",
    "SetPosition",
    "SetSnr",
    "SetTxpower",
    "StartPcap",
    "StopPcap"
  ]
}
```

Get information on the request and response message types for the SetTxpower method.

```
cvd env ls WmediumdService SetTxpower
(Omitted)
{
  "request_type" : "wmediumdserver.SetTxpowerRequest",
  "response_type" : "google.protobuf.Empty"
}
```

Get detailed information for the wmediumdserver.SetTxpowerRequest request message type.

```
cvd env type WmediumdService wmediumdserver.SetTxpowerRequest
(Omitted)
```

```
message SetTxpowerRequest {
  string mac_address = 1;
  int32 tx_power = 2;
}
Send RPC request to the WmediumdService service to modify the Wi-Fi signal strength to the desired level.


cvd env call WmediumdService SetTxpower "{mac_address:'42:00:00:00:00:00', tx_power:1}"
(Omitted)
Rpc succeeded with OK status
{}
```

Here's a full summary of how to **control the simulated environment in a Cuttlefish virtual device**, using both the **REST API** and the **command line (`cvd env`)**, with a sample use case (adjusting Wi-Fi signal strength):

🧭 Overview: What You Can Control

Cuttlefish supports **environmental simulation** via:

Service Name	Purpose
`GnssGrpcProxy`	GPS / geolocation control
`OpenwrtControlService`	Virtual Wi-Fi access point (AP) control
`WmediumdService`	Wireless signal simulation (noise, SNR, power, etc.)
`CasimirControlService`	Simulate external NFC devices

🌐 REST API Usage (`https://localhost:1443`)

🔍 Discover APIs

Action	REST API Call Format
List services	`GET /devices/DEVICE_ID/services`
List service methods	`GET /devices/DEVICE_ID/services/SERVICE_NAME`
Get method info	`GET /devices/DEVICE_ID/services/SERVICE_NAME/METHOD_NAME`
Get request/response field types	`GET /devices/DEVICE_ID/services/SERVICE_NAME/TYPE_NAME/type`
Make an RPC call	`POST /devices/DEVICE_ID/services/SERVICE_NAME/METHOD_NAME` + JSON payload in body

🧪 Example: Adjust Wi-Fi Signal Strength via REST

```bash
# POST request to change TX power
POST https://localhost:1443/devices/cvd-1/services/WmediumdService/SetTxpower
Content-Type: application/json

{
  "mac_address": "42:00:00:00:00:00",
  "tx_power": 1
}
```

Expected response:

```json
{}
```

💻 Command Line Usage (`cvd env`)

🔍 Discover via CLI

Action	Command
List available services	**`cvd env ls`**
List methods for a service	**`cvd env ls WmediumdService`**
Get method info	**`cvd env ls WmediumdService SetTxpower`**
View field types	**`cvd env type WmediumdService wmediumdserver.SetTxpowerRequest`**

🧪 Example: Adjust Wi-Fi Signal Strength via CLI

```bash
cvd env call WmediumdService SetTxpower \
'{mac_address:"42:00:00:00:00:00", tx_power:1}'
```

Expected output:

```csharp
Rpc succeeded with OK status
{}
```

📌 Quick Reference: Example Services & Methods

`WmediumdService` (Wi-Fi environment)

- **`SetTxpower`**: Set signal strength (dBm)
- **`SetSnr`**: Set signal-to-noise ratio
- **`SetPosition`**: Simulate physical position
- **`LoadConfig`**, **`ReloadConfig`**, **`StartPcap`**, **`StopPcap`**

`GnssGrpcProxy` (GPS)

- Can be used to simulate movement or GPS lock

`CasimirControlService` (NFC)

- Simulate external NFC readers/writers

🧠 Pro Tip

For **automation or testing**, you can use:

- **`curl`** or **`httpie`** to interact with the REST API
- Bash scripts calling **`cvd env call`** for reproducible state changes

---

Let me know if you'd like help scripting **movement/GPS**, **interference simulation**, or chaining environment changes into a **test automation pipeline**.

Test connectivity of multiple devices

bookmark_border
Background
Running multiple Cuttlefish virtual devices can be done by executing the following command:

launch_cvd --num_instances=n
When you are running two or more instances using --num_instances flag, instances share the connectivity medium for Wi-Fi and Bluetooth. This setting allows the Cuttlefish instances to discover and connect to each other via Bluetooth and Wi-Fi without additional action.

However, if you are executing multiple instances by using the --base_instance_num flag and multiple launch_cvd invocations, you need to specify wmediumd's vhost server path to share Wi-Fi medium, and also prevent launching multiple VM instances for AP.

Multi-device launch example

The following commands show how to launch two Cuttlefish instances that share the Wi-Fi medium using serial execution of launch_cvd.

launch_cvd

launch_cvd --base_instance_num=2 \
  --vhost_user_mac80211_hwsim=$HOME/cuttlefish_runtime.1/internal/vhost_user_mac80211 \
  --ap_kernel_image="" --ap_rootfs_image="" #Prevent launching multiple VM instances for AP
Bluetooth
The Cuttlefish bluetooth implementation is supported by rootcanal and can be controlled with the Web UI command line console.

In the console, there are several commands that allow for the control of bluetooth on the guest device:

Command Description
list List current devices and phys
add DEVICE_TYPE [ARGS] Create new device of type DEVICE_TYPE
del DEVICE_INDEX Delete a device
add_phy PHY_TYPE Add new phy with PHY_TYPE
del_phy PHY_INDEX Delete a phy
add_device_to_phy DEVICE_INDEX PHY_INDEX Add a device to phy
del_device_from_phy DEVICE_INDEX PHY_INDEX Delete a `device from phy`
add_remote HOSTNAME PORT PHY_TYPE Add a remote Rootcanal as device and register it to phy
Device type and args
beacon [MAC_ADDRESS] [ADVERTISE_INTERVAL]
scripted_beacon [MAC_ADDRESS] [CONFIG_PATH] [EVENTS_PATH]
keyboard [MAC_ADDRESS] [ADVERTISE_INTERVAL]
loopback [MAC_ADDRESS] [ADVERTISE_INTERVAL]
sniffer
Wi-Fi
wmediumd_control
wmediumd_control is a command-line tool that controls Wi-Fi connectivity medium. The tool has the following commands to manipulate Wi-Fi:

Command Description
set_snr MAC_ADDR1 MAC_ADDR2 SNR_VALUE Set SNR value (0~255) between two stations
start_pcap PCAP_FILE_PATH Start capturing all frames transmitted to air and stores it to .pcap file
stop_pcap Stop and finalize current packet capture
list_stations List all stations attached to wmediumd
reload_config [CONFIG_FILE_PATH] Reload wmediumd configuration with CONFIG_FILE_PATH, if CONFIG_FILE_PATH is omitted, reload currently loaded config file
set_position MAC_ADDR X_POSITION Y_POSITION Set position of the station in two-

dimensional space
OpenWRT
If you want to connect to OpenWRT (Access Point), you can use either ssh or the admin webpage.

Connection Type Method
ssh

ssh root@192.168.96.2
Admin webpage https://192.168.96.2

Here's a simple explanation and summary of **testing connectivity between multiple Cuttlefish virtual devices** using Wi-Fi and Bluetooth:

🔧 How to Launch Multiple Cuttlefish Devices

There are two ways:

✅ Easy way (automatic):

```bash
launch_cvd --num_instances=2
```

- This launches 2 Cuttlefish VMs in one go.
- Wi-Fi and Bluetooth sharing is automatically handled.
- You don't need to configure **`wmediumd`** manually.

🛠️ Manual way (with more control):

If you want to launch Cuttlefish devices separately but still let them share Wi-Fi:

1. **Start first device:**

```bash
launch_cvd
```

2. **Start second device (base_instance_num=2):**

```bash
launch_cvd --base_instance_num=2 \
  --vhost_user_mac80211_hwsim=$HOME/cuttlefish_runtime.1/internal/vhost_user_mac80211 \
```

```
--ap_kernel_image="" \
--ap_rootfs_image=""
```

- `vhost_user_mac80211_hwsim`: Shares the Wi-Fi backend (managed by `wmediumd`) with the first instance.
- `ap_kernel_image` and `ap_rootfs_image` are disabled to **avoid running duplicate Wi-Fi APs**.

📶 Testing Bluetooth

Use **Rootcanal**, which simulates Bluetooth on Cuttlefish.

Access Web UI:

Visit: `https://localhost:<web_port>/#/bt-rootcanal-console`

You can run commands like:

Command	Description
`list`	Shows all Bluetooth devices and physical layers
`add beacon XX:XX:XX:XX:XX:XX`	Adds a new beacon device
`add_device_to_phy <dev_id> <phy_id>`	Binds a device to a specific PHY
`add_remote <host> <port> <phy_type>`	Connects to other Rootcanal instances

📡 Testing Wi-Fi

Use `wmediumd_control` to test and simulate different Wi-Fi conditions:

Command	Description
`list_stations`	Lists all Wi-Fi stations (i.e., devices)
`set_snr MAC1 MAC2 <value>`	Adjust signal quality between two MACs (0 = poor, 255 = perfect)
`set_position MAC X Y`	Emulates device distance in a 2D space
`start_pcap`	Start capturing packets
`stop_pcap`	Stop capturing packets

🌐 Connect to Access Point (OpenWRT)

Cuttlefish provides an AP (OpenWRT) that runs at:

- **SSH:** `ssh root@192.168.96.2`
- **Web Admin:** `https://192.168.96.2`

This lets you test real Wi-Fi associations from the Cuttlefish virtual Android devices.

✅ Summary of Steps to Test Multi-Device Connectivity:

1. Launch devices using either `--num_instances` or separate `launch_cvd` with `vhost_user_mac80211_hwsim`.
2. Use `wmediumd_control` to simulate or tweak Wi-Fi environment.
3. Use Rootcanal via the Web UI to simulate Bluetooth devices.
4. Test connection between virtual Android devices by:
   - Scanning available Wi-Fi.
   - Connecting over Bluetooth (via simulated keyboard/beacon/etc.).
5. Optionally test connections to the built-in OpenWRT AP at `192.168.96.2`.

Would you like a working script that launches 3 virtual Cuttlefish instances with full Wi-Fi/Bluetooth simulation support?

Create a Cuttlefish hybrid device

bookmark_border
This page describes how to create a Cuttlefish hybrid device (CHD).

A CHD is a hybrid virtual Cuttlefish device, which runs a physical device's system image instead of a Cuttlefish system image on the device's HALs. This offers the following advantages:

Early development and testing: CHDs allow for early development and testing of next-generation system software before the physical hardware is available.

Scalability: CHDs facilitate easier expansion of development and testing capacity.

To create a CHD, do the following:

Generate the target files for the Cuttlefish device and the physical device using make dist.

Combine the two target files into a CHD target file and generate the CHD images from the CHD target file by running the build_cf_hybrid_device.py script:

```
python3 tools/treble/cuttlefish/build_cf_hybrid_device.py \
  --framework_target_files_zip <physical_device_target_files.zip> \
  --vendor_target_files_zip <cuttlefish_device_target_files.zip> \
  --otatools_zip <cuttlefish_otatools.zip> \
  --target chd \
  --output_dir <output_directory>
```

This script combines the two target files into a CHD target file using merge_target_files.py and generates the CHD images from the CHD target file using img_from_target_files.

To **create a Cuttlefish Hybrid Device (CHD)** — a virtual Cuttlefish device that runs a **physical device's system image** on top of the **Cuttlefish HALs and kernel** — follow these detailed steps:

🔧 What is a CHD?

A **Cuttlefish Hybrid Device (CHD)** enables:

- **Early development** of framework/system-level features **before physical hardware exists**.
- **Virtual testing** using a real device system image.
- **Scalable CI**, as CHDs can be easily launched in parallel on host machines.

✅ Preconditions

You need:

- A Cuttlefish build environment (AOSP checkout with Cuttlefish support).
- A built physical device system image.
- A built Cuttlefish device image.
- Python 3 and the **`build_cf_hybrid_device.py`** script (part of AOSP).

📦 Step-by-step Workflow

1. Build target files for both devices

From your AOSP source tree, build the system images:

```bash
# Build for physical device (e.g., Pixel 7)
lunch aosp_goto-userdebug
make dist -j

# Output: out/dist/<physical_device_target_files>.zip

# Build for Cuttlefish
lunch aosp_cf_x86_64_phone-userdebug
make dist -j

# Output: out/dist/<cuttlefish_target_files>.zip
```

---

2. Build OTA tools (if not already done)

```bash
make otatools -j
# Output: out/host/linux-x86/otatools.zip
```

You can zip the OTA tools manually if needed:

```bash
cd out/host/linux-x86
zip -r ../../dist/otatools.zip otatools
```

---

3. Run the `build_cf_hybrid_device.py` script

Use the provided tool to merge target files and generate CHD images.

```bash
python3 tools/treble/cuttlefish/build_cf_hybrid_device.py \
  --framework_target_files_zip out/dist/physical_device_target_files.zip \
  --vendor_target_files_zip out/dist/cuttlefish_target_files.zip \
  --otatools_zip out/dist/otatools.zip \
  --target chd \
  --output_dir out/chd_output
```

This does the following under the hood:

- Calls **`merge_target_files.py`** to combine framework and vendor target files.
- Calls **`img_from_target_files`** to build the CHD system images.
- Generates final images in **`out/chd_output/`**.

▶️ Launch the CHD image in Cuttlefish

```bash
launch_cvd \
  --system_image=out/chd_output/system.img \
  --data_image=out/chd_output/userdata.img \
  --kernel_path=out/chd_output/kernel \
  --initramfs_path=out/chd_output/ramdisk.img
```

You may include more options like **`--x_res`**, **`--y_res`**, or **`--memory_mb`** as needed.

🧪 Result

You now have a **hybrid Cuttlefish device** running the physical device's system image on Cuttlefish's HALs and kernel — useful for platform validation, debugging, or pre-silicon development.

Would you like a working shell script template for this entire process?

confusing

No worries — let me explain **in simple steps** how to create a **Cuttlefish Hybrid Device (CHD)**.

🧠 What is CHD?

A **CHD** is a **virtual Cuttlefish device** that uses:

- The **system (framework) image** from a **real/physical device** (like Pixel 7)
- The **vendor (HAL + kernel)** image from **Cuttlefish**

👉 This lets you **test new Android versions** on a virtual device **before real hardware is available**.

✅ What You Need

1. **AOSP source tree** with lunch targets for:
    - Your **physical device** (e.g., `aosp_goto`)
    - **Cuttlefish** (e.g., `aosp_cf_x86_64_phone`)
2. `build_cf_hybrid_device.py` script — already inside AOSP.
3. Python 3.

🔲 Step-by-Step Guide

🏗️ Step 1: Build target files for both devices

Open a terminal in your AOSP source directory.

📦 1A. Physical device build

```bash
source build/envsetup.sh
lunch aosp_goto-userdebug
```

```
make dist -j
```

➡️ This creates:
`out/dist/aosp_goto-target_files-*.zip`

📦 1B. Cuttlefish device build

```bash
lunch aosp_cf_x86_64_phone-userdebug
make dist -j
```

➡️ This creates:
`out/dist/aosp_cf_x86_64_phone-target_files-*.zip`

🧰 Step 2: Build OTA tools (if not built)

```bash
make otatools -j
cd out/host/linux-x86/
zip -r ../../../dist/otatools.zip otatools
```

➡️ You now have:
`out/dist/otatools.zip`

🧪 Step 3: Run the hybrid device build script

Now combine the two into a CHD image:

```bash
python3 tools/treble/cuttlefish/build_cf_hybrid_device.py \
  --framework_target_files_zip out/dist/aosp_goto-target_files-*.zip \
  --vendor_target_files_zip out/dist/aosp_cf_x86_64_phone-target_files-*.zip \
  --otatools_zip out/dist/otatools.zip \
  --target chd \
  --output_dir out/chd_output
```

➡️ This gives you:

- **`system.img`**
- **`vendor.img`**
- **`userdata.img`**
- **`boot.img`**

- and more in: **`out/chd_output/`**

🚀 Step 4: Launch the hybrid device

```bash
launch_cvd \
  --system_image=out/chd_output/system.img \
  --userdata_image=out/chd_output/userdata.img
```

You can also add **`--x_res`**, **`--memory_mb`**, etc.

🏁 You're Done!

You've launched a **virtual device** using:

- **Cuttlefish hardware/kernel**
- **Real device system/framework**

Perfect for testing Android builds **before hardware is ready**.

Would you like a complete shell script to automate all of this?