Fuchsia Internals — Volume II

The Build System & Toolchains

Fuchsia Internals Series

***Abstract*—The Fuchsia build is widely regarded as opaque on first contact, principally because of its multi-toolchain model: a single source target may be compiled many times, once per toolchain context, each producing distinct outputs. This reference dissects the entire stack—`jiri` for source, `fx` as workflow wrapper, GN as the meta-build/elaboration engine, Ninja as the executor, the prebuilt Clang and Rust toolchains, in-build code generation (`fidlc`, `cmc`), and Product Assembly that stitches packages into a bootable image. The central goal is to make the *relationships* between these components unambiguous. The treatment is aimed at hardware engineers fluent in RTL elaboration and synthesis: GN is presented as an elaboration pass that instantiates the same design unit in multiple contexts, Ninja as the build runner, and toolchains as cross-compilation cell libraries.**

***Index Terms*—Fuchsia OS, GN, Ninja, build system, toolchain, cross-compilation, FIDL, product assembly, Bazel, packages, Zircon, RBE**

I. Introduction

THE Fuchsia build system confounds newcomers not because any single piece is complex, but because several distinct tools—each with its own vocabulary—are layered into one pipeline, and the seams between them are invisible from a single `fx build` command. The confusion almost always traces back to one idea: *the same target can be built more than once, in different toolchains, producing different binaries.* Once that idea clicks, the rest of the system becomes legible.

This document builds the mental model from the ground up. For a hardware engineer, a precise and load-bearing analogy is available:

- **GN** ("Generate Ninja") is the *elaboration / synthesis* pass. It reads the entire declarative design (the `BUILD.gn` files), resolves every dependency, instantiates each target in each toolchain context that references it, and emits a fully-flattened build graph. GN compiles nothing; it decides *what* will be compiled and *how*.
- **Ninja** is the *build runner* (the place-and-route/bitstream-generation step). It consumes GN's flattened graph (`build.ninja`), runs the actual compiler and linker commands, and handles incremental rebuilds and parallelism.
- **Toolchains** are the *cell libraries / process corners*. Each toolchain is a named bundle of tools (compiler, linker) plus default flags. "Build target `X` in toolchain `Y`" is exactly analogous to "map module `X` onto cell library `Y`"—same RTL, different gates.
- **Product Assembly** is the *final integration / image build*—it takes the compiled pieces and stitches them into a flashable system image, much as a bitstream is assembled from placed-and-routed partitions.

Volume II of the fourteen-volume *Fuchsia Internals* series (I: Graphics & Display Pipeline; II: Build System & Toolchains; III: The Zircon Kernel; IV: Starnix; V: Graphics FIDL APIs in Practice; VI: The Component Framework; VII: Driver Development; VIII: Storage; IX: Power Management; X: Networking; XI: FIDL; XII: Software Delivery & OTA; XIII: Diagnostics & Observability; XIV: Security & Verified Boot). Compiled 2026 from public Fuchsia OS source and documentation at `fuchsia.dev` and `fuchsia.googlesource.com`.

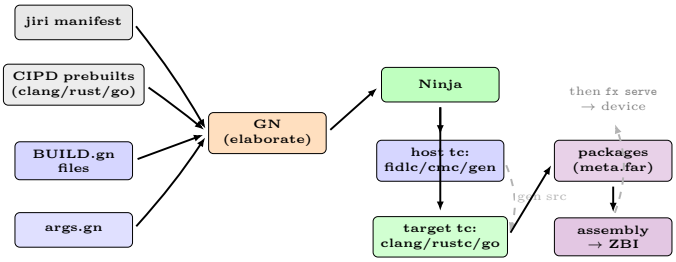

Fig. 1. Relationship map. Solid edges are "produces/feeds"; the dashed edge is the host→target codegen handoff. GN fans the graph across toolchains; Ninja runs host tools that emit source the target toolchain compiles; packages flow into assembly.

TABLE I
Build-tool stack mapped to the RTL/FPGA flow.

Fuchsia tool	Role	RTL/FPGA analogue
jiri	Fetch source + pinned prebuilts	Check out RTL + IP cores
fx	Workflow wrapper / front-end	Vendor tool GUI / TCL driver
GN	Resolve graph, pick toolchains	Elaboration + synthesis
Ninja	Run compile/link commands	Place & route, build runner
Toolchain	Compiler+linker+flags bundle	Cell library / process corner
Variant	Toolchain + instrumentation	Library w/ DFT or timing margin
Assembly	Stitch packages into image	Bitstream assembly

Because the question behind this document is "how do the pieces *relate*?", Fig. 1 maps the whole system on one page: who consumes what, who produces what, and where the toolchain boundary falls. Every later section zooms into one edge of this map.

The remainder of the paper follows the flow of a build. Section II surveys the tool stack and source layout. Section III covers GN fundamentals (targets, labels, the three dependency kinds, configs). Section IV—the centerpiece—dissects the multi-toolchain model in full. Sections V–VIII cover variants, the Clang/Rust compilers, and in-build code generation. Sections IX–X cover packaging and the end-to-end path from `fx set` to a running image. The remaining sections cover the SDK/IDK, the ongoing Bazel migration, build performance, and debugging.

II. The Tool Stack & Source Layout

A. jiri: Source and Prebuilts

A Fuchsia checkout is not a single git repository; it is a *constellation* of repositories plus pinned binary prebuilts,

assembled by `jiri`. The `jiri` tool reads an *integration manifest* (an XML file listing every git project, its remote, and its pinned revision) and lays the tree out under a single root. Throughout the build, that root is referred to with the GN path prefix `//`.

Two categories of content live in the tree:

- **Source**, under paths like `//src`, `//zircon`, `//build`. These are git-managed and built from source.
- **Prebuilts**, under `//prebuilt`, fetched by `jiri` from **CIPD** (Chrome Infrastructure Package Deployment) at versions pinned by the manifest. This is where the Clang toolchain (`//prebuilt/third_party/clang`), the Rust toolchain (`//prebuilt/third_party/rust`), and host tools like `gn` and `ninja` themselves come from. The toolchain you compile *with* is a downloaded, version-locked binary—it is not built from source in an ordinary build.

The pinning is essential: a given Fuchsia revision is reproducible because the exact compiler binary is named by content hash in the manifest. For an RTL engineer this is the equivalent of vendoring a specific synthesis-tool version alongside the RTL so that timing closure is reproducible.

Mechanically, `jiri` marks the checkout root with a `.jiri_root` directory and reads a top-level *integration* manifest (itself a checked-out git project) that `imports` sub-manifests. Each `<project>` element pins a git remote and revision; each `<package>` element pins a CIPD package and version. `jiri update` re-syncs every project and re-fetches every prebuilt to the pinned versions; `jiri snapshot` emits a manifest capturing the exact current state, which is how a build is later reproduced bit-for-bit.

Listing 1. Abridged jiri integration manifest (XML).

```
<manifest>
  <imports>
    <import name="integration"
            remote="https://fuchsia.googlesource.com/integration"/>
  </imports>
  <projects>
    <project name="fuchsia" path="."
             remote="https://fuchsia.googlesource.com/fuchsia"
             revision="a1b2c3..."/>
  </projects>
  <packages>
    <package name="fuchsia/third_party/clang/${platform}"
             path="prebuilt/third_party/clang/${platform}"
             version="git_revision:9f2c8d..."/>
  </packages>
</manifest>
```

B. fx: the Workflow Wrapper

Engineers almost never invoke `gn` or `ninja` directly. The `fx` tool is a thin wrapper that records a chosen build directory and forwards to the underlying tools. The handful of commands that matter:

- `fx set PRODUCT.BOARD` — choose what to build (e.g. `core.x64`). This writes an `args.gn` file into the build directory and runs `gn gen`.
- `fx build` — run Ninja against the configured build directory.
- `fx serve` — start a package server so a running device/emulator can fetch packages over the network.
- `fx test` — build and run tests.

A build optimization flag may be passed to `fx set`: `-debug`, `-balanced`, or `-release`, trading compile time against runtime performance and debuggability [1]. Internally `fx` records the active build directory (`.fx-build-dir`) so subsequent commands need no path argument; `fx` subcommands are themselves small scripts under `//tools/devshell`, and `fx ffx` bridges to the device-facing `ffx` tool. Table II lists the commands an engineer touches daily.

TABLE II
EVERYDAY `fx` SUBCOMMANDS.

Command	Action
`fx set`	Choose product.board; write `args.gn`; run `gn gen`.
`fx args`	Open `args.gn` in an editor; re-gen on save.
`fx build`	Run Ninja on the configured build dir.
`fx serve`	Host the package server (TUF repo).
`fx test`	Build and run host/device tests.
`fx emu` / `qemu`	Boot the image in an emulator.
`fx flash`	Flash the image onto hardware.
`fx ota`	Over-the-air update a running device.
`fx shell`	Open a shell on the target.
`fx log`	Stream device logs.
`fx gn` / `ffx`	Pass through to `gn` / `ffx`.

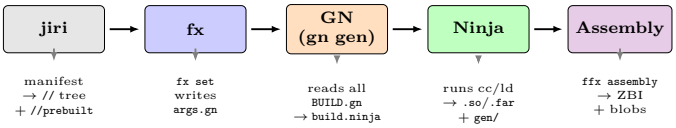

Fig. 2. The Fuchsia build pipeline: each stage's input (top) and product (bottom). GN elaborates; Ninja executes; Assembly integrates.

C. The Build Directory: out/default

`gn gen` populates an output directory, by convention `out/default`. The relevant contents:

- `args.gn` — the human-editable build configuration (product, board, variant selections, feature flags). Editing it and re-running `gn gen` re-elaborates the graph.
- `build.ninja` (and `toolchain.ninja` files) — the flattened build graph GN emitted, one per toolchain instance.
- `host_x64/` — outputs of targets built in the host toolchain (binaries that run on *your* machine: code generators, `fidlc`, `cmc`).
- The target-toolchain outputs (Fuchsia binaries) and shared-library variants such as `x64-shared/`.
- `gen/` — generated source: FIDL bindings, compiled component manifests, and other machine-emitted files, mirrored under the directory of the target that produced them.

The two-phase split (Fig. 2) is the single most important structural fact: `gn gen` is run rarely (only when a `BUILD.gn` or `args.gn` changes—GN auto-emits a rule to regenerate itself), while `ninja` is run on every edit and does the heavy lifting incrementally [2].

A representative pruned view of the directory makes the per-toolchain layout concrete:

Listing 2. Abridged out/default layout (one dir per toolchain).

```
out/default/
  args.gn              # build configuration (editable)
  build.ninja          # default-toolchain build graph
  toolchain.ninja      # rules per toolchain
  obj/                 # default (device) toolchain objects + binaries
    src/foo/bar.cm     # compiled component manifest
  host_x64/            # HOST toolchain outputs (run on your machine)
    fidlc  cmc  gn     # the host tools themselves
  x64-shared/          # shared-library companion toolchain
    libc++.so
  x64-asan/            # asan VARIANT toolchain outputs
  gen/                 # generated source, mirrored by target path
    sdk/fidl/fuchsia.examples/fuchsia.examples.fidl.json
  amber-files/         # package server repository (TUF)
  obj/.../my_pkg/meta.far
```

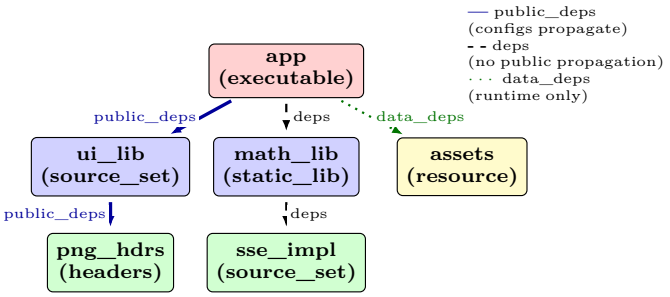

Fig. 3. Dependency kinds. `app` transitively inherits `png_hdrs`' include paths through `ui_lib` (public_deps chain), but *not* `sse_impl`'s, which is hidden behind `math_lib`'s ordinary `deps`.

The mapping is one-to-one: every top-level subdirectory other than `gen/` and the bookkeeping files is the `root_out_dir` of one toolchain instance.

III. GN Fundamentals

GN's input language is declarative. Every directory that contributes to the build holds a `BUILD.gn` file declaring *targets*. GN reads them, builds a dependency graph, and writes Ninja files.

A. Targets

The primitive target types are: `executable()`, `shared_library()`, `static_library()`, `source_set()` (a bag of object files, linked directly into dependents—no intermediate archive), `group()` (a named alias for a set of deps, compiles nothing), `action()` / `action_foreach()` (run an arbitrary script to produce outputs), and `copy()`. Everything else—`fuchsia_package`, `rustc_binary`, `fidl`—is a *template* that ultimately expands into these primitives.

B. Labels

A target is named by a *label*: `//path/to/dir:name`. The shorthand `//path/to/dir` expands to `//path/to/dir:dir`—the target whose name matches its directory. A leading `:name` refers to a target in the current file. As we will see in Section IV, the *full* label syntax has a third, usually-omitted component: the toolchain.

C. The Three Dependency Kinds

This is the first genuine stumbling block. GN distinguishes three dependency lists, and the difference is about *transitivity of public information*:

- `deps` — a normal build-time dependency. The dependency is built first and linked in. Its *public configs* (e.g. include directories it exposes) do *not* propagate further up to *your* dependents.
- `public_deps` — a build-time dependency whose public configs and headers *do* propagate transitively to your dependents. Use this when your own public headers `#include` a header from the dependency, so that anyone using you also transparently gets the include paths.
- `data_deps` — a pure *runtime* dependency. Nothing is linked; the target's outputs (a data file, a helper executable invoked at runtime) are simply made present in the package. No build-time coupling at all.

D. Configs

Compiler and linker flags are not written inline; they are bundled into *config* objects, which are then composed onto targets. A `config()` declares `cflags`, `defines`, `include_dirs`, `ldflags`, etc. A target applies configs through its `configs` list, and—crucially—can add or remove them:

Listing 3. A representative BUILD.gn.

```
config("strict_warnings") {
  cflags = [ "-Wall", "-Werror" ]
}

source_set("math_lib") {
  sources = [ "vec.cc", "mat.cc" ]
  public  = [ "vec.h" ]          # headers visible to dependents
  configs += [ ":strict_warnings" ]
  configs -= [ "//build/config:default_optimize" ]
  configs += [ "//build/config:optimize_speed" ]
}

executable("app") {
  sources      = [ "main.cc" ]
  deps         = [ ":math_lib" ]
  public_deps  = [ "//src/ui:ui_lib" ]
  data_deps    = [ ":assets" ]
}
```

A target's default configs are seeded by its toolchain (Section IV); `configs += / -=` is how a single target opts into or out of global policy.

Config Ordering Matters: Flags are emitted in config-list order, and for many compiler options *last wins*. The toolchain supplies a default list (say `default_optimize`); a target that wants a different optimization level must *remove* the default and *append* its replacement, not merely append—otherwise both `-O0` and `-O2` reach the compiler and the result is order-dependent. This is why Fuchsia code is dotted with the `configs -= [...]` then `configs += [...]` idiom seen above: it is explicit override, the build-system equivalent of a constraint that must supersede a default rather than coexist with it.

E. Build Arguments

`declare_args()` introduces tunable build variables with defaults; the user overrides them in `args.gn`. This is the knob panel of the build:

Listing 4. Declaring and consuming a build argument.

```
declare_args() {
  # Enable the experimental fast path.
  enable_fast_path = false
}

source_set("engine") {
  sources = [ "engine.cc" ]
  if (enable_fast_path) {
    sources += [ "fast.cc" ]
    defines  = [ "FAST_PATH=1" ]
  }
}
```

F. Templates

Templates are GN's macro/abstraction mechanism. A `template("name")` captures a body parameterized by `target_name` and `invoker.<field>`; invoking it expands into one or more primitive targets. The Fuchsia-specific build vocabulary—`fuchsia_package()`, `fuchsia_component()`, `rustc_binary()`, `fidl()`—is entirely templates layered over the primitives [1]. This is why reading a `BUILD.gn` feels high-level while the emitted Ninja graph is low-level: the templates expanded in between. A template body typically begins with `forward_variables_from(invoker, "*")` to pull the caller's arguments into scope, then declares the primitive targets that realize them.

G. Visibility, testonly, and Assertions

Two attributes police the graph. `visibility` is a list of label patterns allowed to depend on a target; a dependency from outside the list is a `gn gen` error, which is how a library prevents accidental coupling (the analogue of a `private` port). `testonly = true` marks a target as test-only; a non-test target depending on it is an error, keeping test code out of production images. `assert(cond, "msg")` fails elaboration early with a readable message—used liberally to reject impossible toolchain/argument combinations before Ninja ever runs.

H. Metadata Collection: How a Package Knows Its Contents

One mechanism deserves special attention because it explains how high-level templates work: *metadata walks*. Any target may attach a `metadata` block of key/value lists. A `generated_file()` target (or a template using `exec_script`/`write_file`) can then perform a *walk*: starting from a root target, GN traverses the dependency graph and collects the named metadata keys from every reachable target into a single list, written to a file at gen time.

This is precisely how `fuchsia_package()` discovers what to put in a package. Each `fuchsia_component()`, resource, and binary in the dependency subtree contributes metadata entries (its manifest path, its blob, its destination path); the package template walks the subtree, aggregates those entries into a *package manifest*, and feeds that to the archiver. The developer never enumerates files manually—the graph does it. For an RTL engineer this is analogous to a tool collecting all `(* keep *)` attributes across a netlist hierarchy into one constraints file.

I. Useful GN Built-ins

A handful of functions appear constantly: `rebase_path()` converts between GN's `//`-rooted paths and the relative paths tools expect; `get_label_info()` extracts a target's output dir, name, or toolchain (essential for cross-toolchain references, as in Fig. 5); `get_target_outputs()` returns the files a target produces; `exec_script()` runs a script *at gen time* (used sparingly—it slows elaboration); and `read_file`/`write_file` do gen-time I/O.

J. The `action()` Primitive in Detail

Every code generator ultimately bottoms out in `action()` (or `action_foreach()`), so it pays to understand it precisely. An action runs a `script` (a Python program, by convention) with `args`, declaring its `sources` (inputs Ninja must watch) and `outputs` (files it promises to produce). The contract is *hermetic*: Ninja assumes the outputs depend only on the declared inputs, so an action that secretly reads an undeclared file will not rebuild when that file changes—the classic "stale build" bug. For inputs that cannot be known until the tool runs (e.g. `#include`s discovered by a compiler), an action emits a `depfile` (Makefile-syntax dependency list) that Ninja reads back to complete the graph.

Listing 5. An action() that runs a host tool.

```
action("generate_table") {
  script  = "//build/gen_table.py"
  sources = [ "table.in" ]
  outputs = [ "$target_gen_dir/table.cc" ]
  depfile = "$target_gen_dir/table.d"
  args = [
    "--in",  rebase_path("table.in", root_build_dir),
    "--out", rebase_path("$target_gen_dir/table.cc", root_build_dir),
    "--depfile", rebase_path(depfile, root_build_dir),
  ]
  # If the generator is a compiled host tool, depend on its
  # host instance and invoke it from its output directory.
  deps = [ "//tools/gen:gen($host_toolchain)" ]
}
```

This hermeticity requirement is not pedantry: it is exactly what makes the action cacheable and remotely executable under RBE (Section XVII). An action with fully-declared inputs and outputs can be shipped to a remote worker, or its result fetched from cache, with confidence that the answer is identical—the build-system analogue of a combinational block with no hidden state.

IV. The Toolchain Model

This is the section the rest of the document builds toward. Internalize one sentence and everything else follows: **in GN, a single target can be elaborated once per toolchain, and each elaboration is a separate instance with its own outputs.**

A. What a Toolchain Is

A GN `toolchain()` is a named definition of *how to turn source into binaries*: it lists the concrete `tool()` commands (`cc`, `cxx`, `link`, `solink`, `stamp`, `copy`, ...) and, through `toolchain_args`, a set of default build-argument values and default configs. Two toolchains might invoke the *same* Clang binary but with a different target triple, different sysroot, and different default flags—and they are, for GN's purposes, entirely distinct compilation environments [3].

The Fuchsia build defines, among others [1], [4]:

- `//build/toolchain:host_x64` — build for the *host* machine (the one running the build).
- `//build/toolchain/fuchsia:x64` — build Fuchsia device binaries for x86-64.
- `//build/toolchain/fuchsia:arm64` — build Fuchsia device binaries for ARM64.
- shared-library and variant toolchains such as `//build/toolchain/fuchsia:x64-asan-shared`.

Concretely, a toolchain is a block of `tool()` definitions. Each `tool` names a Ninja rule and a command template with substitution placeholders (`{{source}}`, `{{output}}`, `{{cflags}}`, ...) that GN fills in per target. A heavily abridged sketch:

Listing 6. Skeleton of a toolchain() definition.

```
toolchain("fuchsia_x64") {
  tool("cc") {
    command = "$clang_path/clang {{cflags}} {{cflags_c}} " +
              "-c {{source}} -o {{output}}"
    outputs = [ "{{source_out_dir}}/{{source_name_part}}.o" ]
  }
  tool("cxx") { command = "$clang_path/clang++ ... -c {{source}} ..." }
  tool("link") {
    command = "$clang_path/clang++ {{ldflags}} {{inputs}} " +
              "{{solibs}} {{libs}} -o {{output}}"
  }
  tool("solink") { /* shared library link */ }
  tool("stamp") { command = "touch {{output}}" }

  # Defaults injected into EVERY target built here:
  toolchain_args = {
    current_cpu = "x64"
    current_os  = "fuchsia"
  }
}
```

Two toolchains can share the same `clang` binary yet differ only in `toolchain_args` and default configs (sysroot, target triple); to GN they are still wholly separate compilation contexts.

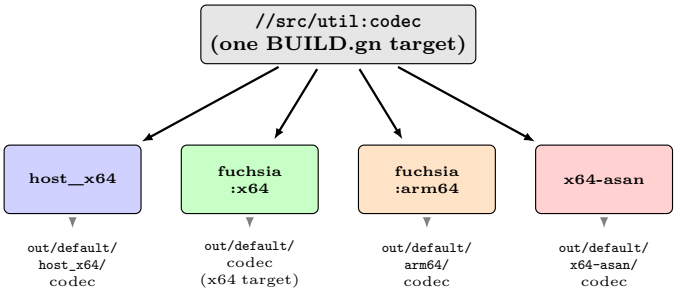

Fig. 4. The same target `//src/util:codec` fanned out across four toolchains. GN elaborates four independent instances; Ninja builds four distinct binaries into four output directories.

B. The Default Toolchain and the Buildconfig

GN bootstraps in exactly one toolchain—the *default*—chosen by a call to `set_default_toolchain()` in the build's *buildconfig* file (named by `.gn` at the source root). For Fuchsia the default is the device target toolchain, which is why an unqualified `fx build //foo:bar` builds the *device* binary. A subtlety that trips people: certain things (e.g. the top-level `//:default` target, some `generated_file` collections) are only evaluated in the default toolchain, so duplicating them in another toolchain is a no-op rather than an error.

C. Why an `x64-shared` Toolchain Exists

A practical wrinkle explains the `x64-shared` output directory. Shared libraries are expensive to rebuild and are ABI-compatible across many variants. Fuchsia therefore builds shared libraries in a dedicated *companion* toolchain (`x64-shared`) so that an `asan` executable and a plain executable can *link the same* `libc++.so` instead of each rebuilding it. This is a pure build-efficiency move, but it is why a single `gn gen` produces a top-level dir, a `host_x64/` dir, an `x64-shared/` dir, and per-variant dirs—each is a toolchain's `root_out_dir`.

D. The Label-with-Toolchain Syntax

The full label form is:

```
//path/to/dir:name(//path/to/toolchain:tc)
```

The parenthesized suffix is the *toolchain in which to build that target*. When you write a plain label `//foo:bar`, GN silently expands it to `//foo:bar($current_toolchain)`—"the same target, in whatever toolchain I am currently being elaborated in." That implicit expansion is why the toolchain is invisible most of the time, and why it is so disorienting when it suddenly matters.

The practical upshot, from the command line, is that these are all ways to build the host `zbi` tool [4]:

Listing 7. Equivalent ways to name one toolchain instance.

```
fx build --host //zircon/tools/zbi
fx build --toolchain=//build/toolchain:host_x64 //zircon/tools/zbi
fx build '//zircon/tools/zbi(//build/toolchain:host_x64)'
fx build host_x64/zbi
```

The last form names the output path directly: a target built in `host_x64` lands under `out/default/host_x64/`.

E. The Key Insight: One Target, Many Instances

Figure 4 is the picture to commit to memory. One `BUILD.gn` target, referenced from four toolchain contexts, becomes four binaries. `//src/util:codec(host_x64)` and `//src/util:codec(//build/toolchain/fuchsia:x64)` share *source* but are otherwise unrelated build nodes: different compiler flags, different sysroot, different output directory, no shared object files. For the RTL engineer: this is the same Verilog module instantiated against two different cell libraries, yielding two different gate-level netlists.

TABLE III
Built-in GN toolchain/context variables.

Variable	Meaning
`current_toolchain`	Label of the toolchain elaborating this file right now.
`default_toolchain`	The toolchain GN starts in (set by `set_default_toolchain`); for Fuchsia, the device target toolchain.
`host_toolchain`	Label of the toolchain that targets the build host.
`target_toolchain`	Label of the primary device toolchain.
`current_cpu`	CPU of the current toolchain (`x64`, `arm64`).
`current_os`	OS of the current toolchain (`fuchsia`, `linux`, `mac`).
`target_cpu` / `target_os`	CPU/OS of the *final product* being built.
`root_out_dir`	Output directory for the current toolchain.
`target_gen_dir`	Generated-file directory mirroring this BUILD.gn.

F. Built-in Toolchain Variables

Inside a `BUILD.gn`, GN exposes variables that tell you which context you are currently being elaborated in, so the *same file* can behave differently per toolchain (Table III). The common idiom:

Listing 8. A file that behaves differently per toolchain.

```
if (current_toolchain == host_toolchain) {
  # Only define the code generator in the host build.
  executable("gen_tool") { sources = [ "gen.cc" ] }
} else {
  # On the target, consume what the host tool produced.
  source_set("generated") { sources = [ "$target_gen_dir/out.cc" ] }
}
```

G. Why This Exists: Crossing the Host/Target Boundary

The multi-toolchain model is not academic; it is how Fuchsia cross-compiles cleanly in a single `gn gen`. Consider the universal pattern: a *code generator* must run on the host (it is a program executed during the build), but the code it *emits* must be compiled for the device. Both live in the same repository and the same build invocation. GN expresses this by having the target-toolchain instance depend on the host-toolchain instance of the tool:

Listing 9. A cross-toolchain dependency.

```
# Built in the TARGET toolchain (device binary):
source_set("app_messages") {
  # Inputs are produced by running the generator ON THE HOST.
  sources = [ "$target_gen_dir/messages.cc" ]
  deps    = [ ":run_generator" ]
}

action("run_generator") {
  # Depend on the HOST instance of the tool, regardless of
  # which target toolchain we are currently in.
  deps    = [ "//tools/gen:gen($host_toolchain)" ]
  tool    = get_label_info(
              "//tools/gen:gen($host_toolchain)", "root_out_dir") +
            "/gen"
  outputs = [ "$target_gen_dir/messages.cc" ]
```

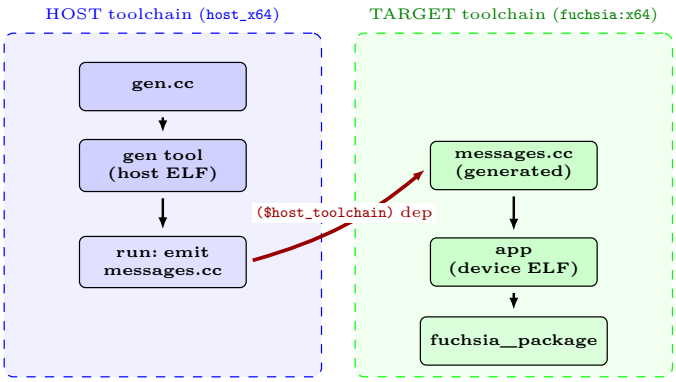

Fig. 5. Cross-toolchain dependency. The device-side `app` needs generated source; the generator runs in the host lane. The red edge is a dependency on the host instance, written `:gen($host_toolchain)`, crossing the toolchain boundary in one `gn gen`.

```
  script  = "//build/run.py"
}
```

Figure 5 shows the boundary crossing explicitly. The host lane builds and runs the generator; the target lane compiles its output into a device binary. The arrow between lanes is the `($host_toolchain)` suffix doing its job. Every FIDL library in Fuchsia is an instance of exactly this pattern (Section VIII).

H. Toolchain Context Propagates Down Dependencies

A point that resolves much confusion: when target A (being elaborated in toolchain T) lists `deps = [ ":B" ]`, the unqualified `:B` is built in *the same toolchain T*. Toolchain context flows down the dependency graph by default—an entire subtree elaborates in whatever toolchain its root was entered in. You only write an explicit `(...)` suffix at the precise points where you want to *cross* into a different toolchain (the host-tool edge). This is why a device executable's hundreds of transitive deps need no suffixes: they all inherit the device toolchain; only the lone code-generator edge carries `($host_toolchain)`.

I. Seeing the Two Instances

The fastest way to make the model tangible is to ask GN to describe the same target in two toolchains and observe that the answers differ:

Listing 10. The same label, two toolchains, two descriptions.

```
# Device instance: Fuchsia triple, sysroot, device out dir.
fx gn desc out/default '//src/util:codec'
# Host instance: host triple, host libc, host_x64/ out dir.
fx gn desc out/default '//src/util:codec(//build/toolchain:host_x64)'
```

The `outputs`, `cflags` (target triple, sysroot), and `toolchain` fields differ between the two; the `sources` are identical. That diff *is* the multi-toolchain model, made visible.

J. How `gn gen` Elaborates

Internally, `gn gen` starts in the default toolchain at the root and walks dependencies. Whenever it encounters a label carrying a different toolchain suffix, it *also* loads and evaluates the relevant `BUILD.gn` files in that toolchain—each `BUILD.gn` is re-executed once per toolchain that references targets in it, with the context variables (`current_toolchain`, `current_cpu`, `is_kernel`, ...) set accordingly. The result is a set of per-toolchain build graphs, written as separate Ninja files and tied together by the cross-toolchain edges. Elaboration terminates when no new (target, toolchain) pair remains to evaluate. This re-evaluation per toolchain is precisely why a `BUILD.gn` can branch on `current_toolchain` and produce different targets in different contexts—it is literally run again each time.

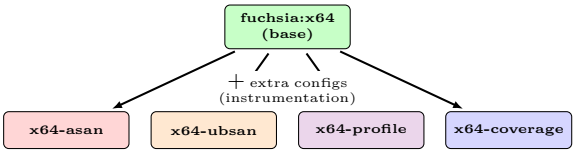

Fig. 6. Variant toolchains derive from a base toolchain by adding instrumentation configs. Each is a full `toolchain()` with its own output directory.

K. Output Directory Layout Recap

Because each toolchain owns a `root_out_dir`, the on-disk layout directly mirrors the toolchain set: the default device toolchain writes to the top of `out/default`; `host_x64/` holds host tools; `x64-shared/` holds shared-library-toolchain outputs; and each variant gets a subdirectory such as `x64-asan/`. If you ever wonder "where did my binary go?", the answer is always "in the output directory of the toolchain it was built in."

V. Toolchain Variants

A *variant* is a toolchain *derived* from a base toolchain by layering extra configs—typically instrumentation. Because a variant is just another toolchain, everything from Section IV applies unchanged: a variant has its own `root_out_dir`, and a target built in a variant is a separate instance [5].

The standard variants are:

- `asan` / `ubsan` — Clang AddressSanitizer / UndefinedBehaviorSanitizer; the combined `asan-ubsan` applies both.
- `coverage` — instrumentation-based profiling for code-coverage collection.
- `profile` — instrumentation for profile-guided optimization (PGO).

Which targets get which variant is chosen by the `select_variant` build argument in `args.gn`. It holds an ordered list of *variant selectors*; each selector names a variant and optionally restricts which targets it applies to (by name, by directory, by output type). Selectors are tried in order and the first match wins. The build system then defines linkable targets in the appropriate variant toolchain automatically.

A variant build links the matching instrumented runtime—for `asan`, the AddressSanitizer runtime shipped in the prebuilt Clang. One constraint is structural: components that must run before runtimes are available (notably code that runs while *booting*, in the ZBI) cannot be instrumented. The build tags such variants `instrumented`, and the bootloader/early-boot assembly excludes them [5].

VI. The Compiler Toolchains: Clang & Rust

A. Clang and the Sysroot

The C/C++ toolchain is a prebuilt Clang under `//prebuilt/third_party/clang`, fetched from CIPD at a pinned revision. Device code is cross-compiled to a Fuchsia target triple—`x86_64-unknown-fuchsia` or `aarch64-unknown-fuchsia`—against the **Fuchsia sysroot**: the set of headers and the C runtime library (provided by the

TABLE IV
COMMON TARGET TRIPLES AND THEIR TOOLCHAINS.

Triple	CPU/OS	Toolchain label
`x86_64-unknown-fuchsia`	x64 / fuchsia	`fuchsia:x64`
`aarch64-unknown-fuchsia`	arm64 / fuchsia	`fuchsia:arm64`
`x86_64-unknown-linux`	x64 / linux	`host_x64`
`aarch64-unknown-linux`	arm64 / linux	`linux_arm64`

TABLE V
LANGUAGE AND CODEGEN TOOLCHAIN CONTEXTS.

Context	Prebuilt	Builds
C/C++	`clang`	device/host/kernel ELF
Rust	`rust`	`rustc_*` crates
Go	`go`	`go_*` (GOOS=fuchsia)
`fidling`	`fidlc`/fidlgen	runs codegen actions only

Fuchsia C library, the userspace face of Zircon's syscalls) that define the device ABI. The sysroot is what makes "compile for Fuchsia" mean something concrete to Clang; it is supplied by the target toolchain's default configs, not written into each `BUILD.gn`.

B. Rust

The Rust toolchain is likewise a pinned prebuilt under `//prebuilt/third_party/rust`. Rust code is not written with raw `rustc` invocations; the GN templates `rustc_binary()` and `rustc_library()` wrap it, handling crate naming, edition, features, and—critically—the dependency edges to other crates. Third-party crates are *vendored* (checked into the tree, not fetched at build time) under `//third_party/rust_crates`, so a Rust build, like everything else, is hermetic and reproducible from the pinned tree.

C. Go

Fuchsia also carries a Go toolchain—a pinned prebuilt under `//prebuilt/third_party/go` with GN templates `go_library()`, `go_binary()`, and `go_test()` wiring it into the build [13]. Because GN has no native Go support, the integration defines a dedicated *Go toolchain context* that drives the Go compiler with a Fuchsia-aware `GOROOT`, `GOOS=fuchsia`, the matching `GOARCH`, and the `fuchsia` build tag. Go reached the system the same way every language does: cgo against the Fuchsia C library plus generated FIDL Go bindings (the `_go` backend of Section VIII). Historically Go implemented components such as the original network stack and a number of host tools; much has since migrated to Rust, but the Go toolchain remains, and host-side Go tools are still built with it. For our purposes the salient point is structural: Go is *another GN toolchain context*, reinforcing the central model—each language is a toolchain, and a target compiles in whichever toolchain its language demands.

D. Runtime Libraries

Cross-compiling for Fuchsia pulls in a specific runtime stack, all from the prebuilt Clang: the C library (Fuchsia's libc, with the Scudo hardened allocator), the C++ standard library `libc++` (shipped as `libc++.so`, built in the shared toolchain as noted in Section IV), the compiler builtins, and—under a variant—the sanitizer runtimes. These are not linked by accident; the target toolchain's default configs (under `//build/config/fuchsia`) add the sysroot, the runtime search paths, and the default `-fPIE`/`-fPIC` policy. A Fuchsia executable is position-independent and dynamically links `libc++.so` and the C runtime as package blobs, which is why those libraries appear as `data_deps`/loadable content of components rather than being statically baked in.

E. C++/Rust Interop

The two languages meet at FIDL. A FIDL library generates both a C++ binding and a Rust binding from the *same* interface definition; a Rust component and a C++ component can then speak the same wire protocol with no hand-written glue. Driver bind rules (the `bind` compiler) provide a parallel mechanism for matching drivers to hardware nodes. Both are code-generation steps, the subject of the next section. Dart and (legacy) Go toolchains exist on the same pattern—a pinned prebuilt plus GN templates—though C++ and Rust dominate the platform.

VII. ZIRCON & KERNEL TOOLCHAINS

Historically Zircon (the kernel and core libraries) had its own separate build (informally "ZN"), bolted to the userspace ("GN") build through a generated bridge. That split is gone: today the kernel builds in the *same* `gn gen` as everything else, simply in *different toolchains*. This is the multi-toolchain model paying off—rather than a second build system, the kernel is just code compiled in kernel-flavored toolchains.

Kernel code is special in ways that map directly onto toolchain differences: it must not use the userspace sysroot or libc, it runs before virtual memory and the C runtime are fully up, and parts of it (the very early *physical-memory* boot shim) run in a constrained machine mode. Fuchsia therefore defines dedicated toolchains—a kernel toolchain (with `is_kernel` true and a freestanding, no-libc config), a `phys` toolchain for early boot, a multiboot/x86 32-bit entry toolchain, and a UEFI bootloader toolchain. Each is a `toolchain()` with its own tools, default configs, and `root_out_dir`, exactly like the userspace toolchains.

Listing 11. Behaving differently in the kernel toolchain.

```
source_set("ring_buffer") {
  sources = [ "ring.cc" ]
  if (is_kernel) {
    # No libc here; use the in-kernel allocator + headers.
    deps = [ "//zircon/kernel/lib/heap" ]
  } else {
    deps = [ "//sdk/lib/fit" ]
  }
}
```

The same `is_kernel`/`current_toolchain` tests from Section IV let one `BUILD.gn` serve both worlds. The kernel image produced by these toolchains is then handed to assembly as the kernel payload of the ZBI (Section IX). The conceptual win is large: there is exactly one elaboration model for the entire OS, kernel included, and "the kernel build" is not a separate tool but a set of toolchain contexts.

VIII. CODE GENERATION IN THE BUILD

A large fraction of Fuchsia's source is generated during the build. The dominant case is FIDL, and it is the canonical instance of the cross-toolchain pattern from Section IV.

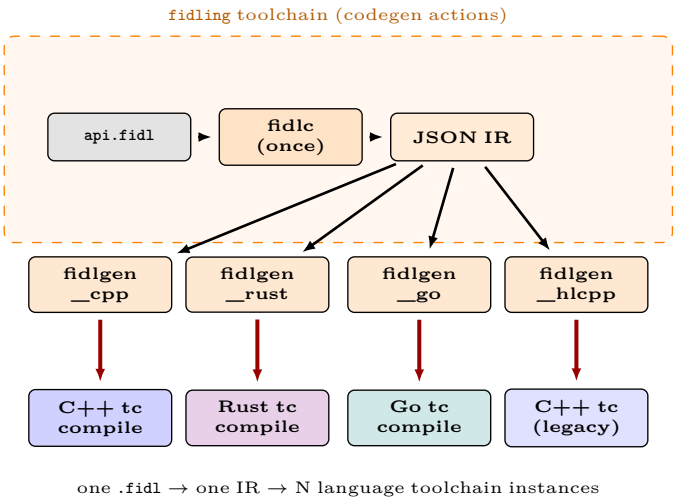

Fig. 7. FIDL fan-out: `fidlc` runs *once* in the dedicated `fidling` toolchain to emit the JSON IR; each `fidlgen` backend consumes it and its output is compiled in that language's own toolchain. This is the multi-toolchain insight of Fig. 4 made concrete—one logical library, many toolchain instances.

A. The FIDL Two-Stage Compiler

FIDL compilation is deliberately split [6]:

1) **Frontend: `fidlc`.** Consumes the `.fidl` files of a library (plus all transitive dependency libraries), performs semantic analysis, and emits a language-agnostic **JSON IR** file, `${target_name}.fidl.json`. `fidlc` is a host tool.
2) **Backends: `fidlgen_*`.** Each backend (`fidlgen_cpp`, `fidlgen_rust`, `fidlgen_hlcpp`, `fidlgen_go`, ...) consumes the JSON IR and emits bindings in one language. These too are host tools.

The emitted bindings (`.cc`/`.h`, `.rs`, `.go`) are then compiled *in the appropriate language/target toolchain* as ordinary source and linked into the component that uses them.

Here the build does something worth dwelling on, because it is the cleanest real-world instance of the Section IV insight. The `fidlc` and `fidlgen` *actions* do not run in the host or the device toolchain—they run in a dedicated GN toolchain named `fidling` (the `//build/fidl` "fidling" context), a toolchain whose only job is to run these code-generation scripts via `action()` targets [6]. The frontend runs *once* in `fidling` to produce the IR; that single IR then feeds every language backend. Each backend's emitted source is finally compiled in *its own* language toolchain—C++ in the device (or host) C++ toolchain, Rust in the Rust toolchain, Go in the Go toolchain.

So a single `fidl("my.library")` declaration fans out across toolchains exactly as Fig. 4 promised in the abstract: one logical library, multiple toolchain instances, distinct outputs. The `fidl()` template exposes per-language sub-targets (`my.library_cpp`, `my.library_rust`, `my.library_hlcpp`, `my.library_go`); a consumer depends on whichever it needs, and that language's bindings are generated and compiled lazily—an unused backend never runs.

B. A Worked FIDL Target

Consider a one-method library. The `fidl()` template, given the `.fidl` below, generates a family of per-language sub-targets; a consumer depends on whichever language it needs.

TABLE VI
PER-LANGUAGE SUB-TARGETS GENERATED BY `fidl()`.

Sub-target suffix	Binding produced
`_cpp`	New C++ (natural + wire domain objects)
`_hlcpp`	High-level C++ (legacy)
`_rust` / `-rustc`	Rust crate binding
`_go`	Go binding (built in the Go toolchain)
`_llcpp`	Low-level/wire C++ (legacy alias)
`(no suffix)`	The `fidlc` JSON IR target (in `fidling`)

Listing 12. echo.fidl — a minimal library.

```
library fuchsia.examples;

@discoverable
closed protocol Echo {
    strict EchoString(struct { value string:64; })
        -> (struct { response string:64; });
};
```

Listing 13. Its BUILD.gn and a C++ consumer.

```
import("//build/fidl/fidl.gni")

fidl("fuchsia.examples") {     # runs fidlc + fidlgen backends
  sources = [ "echo.fidl" ]
}

executable("server") {
  sources = [ "server.cc" ]
  # Depend on the C++ (natural+wire) binding sub-target:
  deps = [ ":fuchsia.examples_cpp" ]
}
```

Because bindings are generated only when depended upon, an unused language backend costs nothing. The IR target (no suffix) is the shared host-side product; each language sub-target is a host `fidlgen` run plus a target-toolchain `source_set` of the emitted code.

C. Other Generators

The same host-emit/target-compile shape recurs throughout: the `bind` compiler turns driver bind rules into matchable bytecode that the driver manager evaluates against device-tree node properties; `cmc` compiles component manifests (Section IX); Banjo (the legacy driver-transport IDL, largely superseded by FIDL-at-the-driver-boundary) generated C headers the same way. In every case the generator is a host target and its output is target source. The lesson generalizes: *any* time Fuchsia needs machine-generated source, it expresses the generator as a host-toolchain target and consumes its output in the target toolchain—the cross-toolchain edge is the universal idiom.

IX. Packages, Components & Assembly

Compilation produces ELF binaries; those binaries reach a device only after being wrapped into *packages* and assembled into a system *image*.

A. Components and Manifests

A **component** is the unit of software execution on Fuchsia, declared by a **component manifest**. Authors write `.cml` (a JSON5 dialect); the `cmc` tool validates and compiles it to a binary `.cm` that is stored in the component's package. The `fuchsia_component()` GN template runs `cmc` and bundles the resulting `.cm` with the component's executable.

B. Packages, FARs, Blobs, and Merkle Roots

A **Fuchsia package** is not a single file but a *tree of content-addressed blobs*. Its structure [7], [8]:

- Every file in the package is a **blob**, named by the hash of its contents under the **Fuchsia Merkle Root** algorithm. Identical content yields an identical name—automatic deduplication across all packages on the system.
- The root blob is `meta.far`, a **FAR** (Fuchsia ARchive—a simple `tar`-like path→contents container). It holds the package metadata, including `meta/contents`, which maps each in-package path to the Merkle root of its blob, plus the compiled `.cm` manifests.
- The content hash of `meta.far` itself is the **package hash**: the single identifier that names the whole package.

The `fuchsia_package()` template gathers components, binaries, and resources (via the metadata walk of Section III), computes blobs, and produces the `meta.far`. Because addressing is by content hash, software *delivery* and *update* reduce to fetching whichever blobs are not already present. Inside `meta.far`, beyond `meta/contents`, sit `meta/package` (package name and version) and an *ABI revision* stamp that ties the package to a platform API level—the resolver refuses to run a package whose ABI revision the running platform does not support, preventing silent ABI drift.

C. Delivery: the Package Resolver and TUF

Packages are served and updated through a content repository secured by **TUF** (The Update Framework). `fx serve` hosts a TUF repository (the `amber-files/` directory in the build output) advertising signed metadata over the network. On the device, the *package resolver* fetches a package by name, validates the TUF metadata chain, retrieves the `meta.far`, reads `meta/contents`, and then fetches only the missing blobs by Merkle root—blobs already present (shared with another package) are never re-downloaded. This content-addressed deduplication is what makes OTA updates small: only changed blobs move.

Packages are named by URL: `fuchsia-pkg://fuchsia.com/echo` addresses package `echo` in the `fuchsia.com` repository, and a component within it is `fuchsia-pkg://fuchsia.com/echo#meta/echo.cm` (the fragment is the in-package path of the compiled manifest). A package may also declare **subpackages**—other packages it references by name and pinned hash—so a parent can compose children hermetically without flattening their blobs, the delivery-layer analogue of an IP block instantiating a pinned sub-core.

D. Product Assembly

Individual packages are combined into a bootable system by **Product Assembly**, driven by `ffx assembly`. Its inputs [9]:

- **Assembly Input Bundles (AIBs)** — self-contained directories of packages, drivers, and config, each with a manifest, grouping platform pieces for inclusion together.
- a **product configuration** (what product-level software and settings to include) and a **board configuration** (the hardware target).

The `ffx assembly product` invocation takes these as explicit arguments—`-product`, `-board-config`, `-input-bundles-dir`, plus `-outdir`/`-gendir` for results. The product configuration is a JSON document selecting platform features and supplying their settings; assembly validates it against a schema and refuses contradictory requests:

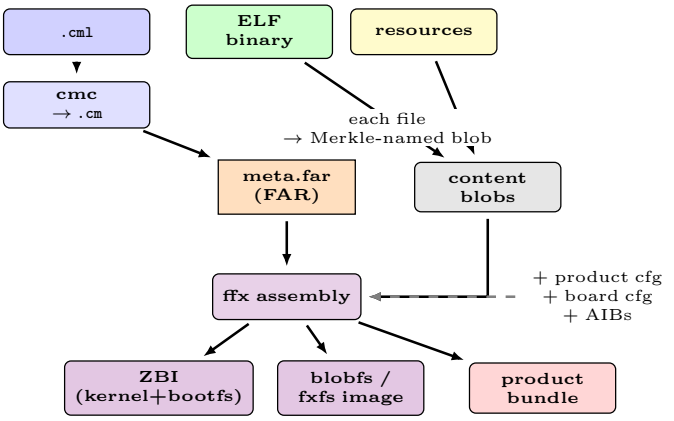

Fig. 8. From source to image. `fuchsia_package` builds `meta.far` + blobs; `ffx assembly` combines packages with product/board config and Assembly Input Bundles to emit the ZBI, the blob filesystem image, and the product bundle.

Listing 14. Sketch of a product configuration (JSON).

```
{
  "platform": {
    "build_type": "eng",
    "connectivity": { "network": { "netstack_version": "ns3" } },
    "ui": { "supported_input_devices": [ "touchscreen" ] }
  },
  "product": {
    "packages": { "base": [ { "manifest": ".../echo/package_manifest.json" }
      ] }
  }
}
```

An AIB is similarly self-describing: a directory with an `assembly_config.json` manifest naming its packages, drivers, kernel args, and config-data. Platform teams ship AIBs; product owners pick which to include. This is the final expression of the build's separation of concerns: compilation produced blobs, and assembly is a *pure data* step selecting and arranging already-compiled artifacts—no compiler runs during `ffx assembly`. Assembly proceeds in two conceptual phases: *Product Assembly* selects and combines platform and product pieces into the full set of things to ship; *Image Assembly* packs those into deliverable images—the base/system package, the **ZBI** (Zircon Boot Image: kernel + bootfs + boot arguments), and a block image such as FVM/blobfs (or fxfs). The bundled result that can be flashed, OTA-updated, or booted in an emulator is a **product bundle**.

The **ZBI** deserves emphasis because it is the bootstrapping floor. It contains the Zircon kernel plus a *bootfs*: a minimal in-memory filesystem with just enough drivers and components to bring up storage, networking, and the package resolver. Drivers needed before a filesystem exists (storage, the bus that hosts it) must live in the ZBI; everything else loads later from packages on the blob volume. The block image holds those packages: historically **blobfs** (a flat, Merkle-verified blob store) inside an **FVM** (Fuchsia Volume Manager) partition, increasingly **fxfs** (the newer filesystem). Verified boot extends the Merkle chain from the bootloader through the ZBI into every blob, so tampering anywhere is detected.

E. Build Types: eng, userdebug, user

A product's *build type* sets the security/debuggability posture and is chosen at assembly time. `eng` is the permissive engineering

TABLE VII
BUILD/ASSEMBLY ARTIFACT TYPES.

Artifact	Role
`.cml`	Component manifest source (JSON5), authored.
`.cm`	Compiled manifest (binary), emitted by `cmc`.
`.far`	Fuchsia archive; `meta.far` is the package root.
blob	Content-addressed file, Merkle-named.
package hash	Merkle root of `meta.far`; names a package.
`.zbi`	Zircon Boot Image: kernel + bootfs + bootargs.
blobfs/fxfs	On-disk filesystem image holding all blobs.
product bundle	Flashable/emulatable bundle of all images.

build (root shell, full logging, eng-only tools); `user` is the locked-down shipping build (no ambient shell, restricted introspection); `userdebug` sits between them. The same compiled binaries can be assembled into different build types—the difference is which components and capabilities the platform configuration admits, not a recompile. This separation of *what is compiled* from *what is assembled* is exactly the GN/assembly division of labor in miniature.

X. From `fx set` to a Running Image

The pieces now assemble into one end-to-end story (Fig. 9):

1) **Configure.** `fx set core.x64 -with //some:target` writes `args.gn` (product `core`, board `x64`, plus the extra target) and runs `gn gen out/default`.
2) **Elaborate.** GN reads every `BUILD.gn`, expands templates, resolves toolchains—instantiating host tools in `host_x64` and device code in `fuchsia:x64`—and emits `build.ninja` plus per-toolchain Ninja files.
3) **Build.** `fx build` runs Ninja. Dependency order forces *host tools first* (e.g. `fidlc`, `cmc`, code generators); those tools run to emit generated source; device code then compiles and links; `fuchsia_package` targets produce `meta.far` + blobs.
4) **Assemble.** `ffx assembly` combines packages with product/board config and AIBs into the ZBI and filesystem images (the product bundle).
5) **Run.** `fx serve` hosts a package server; the device or emulator boots the image and fetches packages on demand; `fx ota`/`fx reboot` or the emulator brings the system up.

XI. A Complete Worked Example

Abstractions land better against a concrete artifact. This section builds one small component—an `echo` server—from nothing, naming every file and every toolchain it touches. The point is not the program; it is to see all of the preceding machinery cooperate on a single buildable unit.

A. The Source Files

Four files, in one directory `//src/echo`: an interface, an implementation, a manifest, and a build file.

Listing 15. //src/echo/echo.fidl — the interface.

```
library fuchsia.echo;

@discoverable
closed protocol Echo {
    strict EchoString(struct { value string:64; })
        -> (struct { response string:64; });
};
```

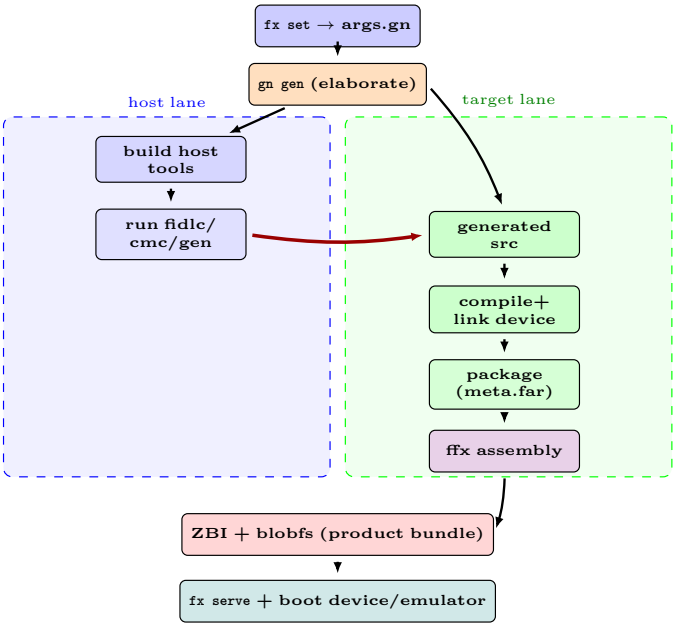

Fig. 9. End-to-end build with host and target lanes. One `gn gen` elaborates both lanes; Ninja sequences host tools → codegen → device compile → package → assembly.

Listing 16. //src/echo/main.cc — the implementation (sketch).

```
#include <fidl/fuchsia.echo/cpp/wire.h>      // generated binding
#include <lib/async-loop/cpp/loop.h>
// ... server implements EchoString() by echoing ‘value‘ back ...
int main() { /* serve fuchsia.echo.Echo on the outgoing dir */ }
```

Listing 17. //src/echo/meta/echo.cml — the component manifest.

```
{
    program: {
        runner: "elf",
        binary: "bin/echo_server",
    },
    capabilities: [ { protocol: "fuchsia.echo.Echo" } ],
    expose: [ { protocol: "fuchsia.echo.Echo", from: "self" } ],
}
```

Listing 18. //src/echo/BUILD.gn — ties it all together.

```
import("//build/components.gni")
import("//build/fidl/fidl.gni")

fidl("fuchsia.echo") {            # -> fidlc in ‘fidling‘, then backends
  sources = [ "echo.fidl" ]
}

executable("bin") {
  output_name = "echo_server"
  sources     = [ "main.cc" ]
  deps = [
    ":fuchsia.echo_cpp",          # C++ binding sub-target
    "//sdk/lib/async-loop:async-loop-cpp",
  ]
}

fuchsia_component("component") {  # runs cmc on the .cml
  component_name = "echo"
  manifest = "meta/echo.cml"
  deps     = [ ":bin" ]
}

fuchsia_package("echo") {         # gathers blobs -> meta.far
  deps = [ ":component" ]
}
```

B. What Each Toolchain Does

Trace the build by toolchain, and the whole document collapses into one example:

1) **fidling:** `fidlc` compiles `echo.fidl` to JSON IR; `fidlgen_cpp` emits the C++ wire/natural binding source. (Had a Rust or Go consumer existed, those backends would also run—each feeding its own toolchain.)
2) **host_x64:** the `cmc` tool (already built) compiles `echo.cml` to a binary `.cm`.
3) **fuchsia:x64** (the default/target toolchain): the generated binding and `main.cc` compile and link into the device ELF `echo_server`, against the Fuchsia sysroot and `libc++.so` from `x64-shared`.
4) back in the default toolchain: `fuchsia_package` performs a *metadata walk* over `:component` and `:bin`, collects the ELF, the `.cm`, and the binding's runtime needs, computes Merkle-named blobs, and writes `meta.far`.

C. The Commands

Listing 19. Building and running the example.

```
# 1. Configure: choose product+board, add our package.
fx set core.x64 --with //src/echo
# 2. Build everything (host tools, codegen, device, package).
fx build
# 3. Inspect: confirm the package target's resolved deps.
fx gn desc out/default //src/echo:echo
# 4. Serve packages and run the component on a device/emulator.
fx serve &
ffx component run /core/ffx-laboratory:echo \
    fuchsia-pkg://fuchsia.com/echo#meta/echo.cm
```

A single `fx build` drove four toolchains, two code generators, a cross-compile, and a package archive—and nothing about it required the engineer to think about toolchains explicitly, because the templates and the default-toolchain rules placed each target in the right context automatically. The machinery only becomes visible (and the `($host_toolchain)` suffix only becomes necessary) when you step outside the templates, as the next section's pitfalls show.

XII. Common Pitfalls & FAQ

Almost every newcomer stumble traces to the multi-toolchain model. Collected here are the recurring ones, each with the underlying cause.

"My host test/tool builds for the device (wrong libc, link errors)." A dependency on a host target omitted the toolchain suffix. Write `:tool($host_toolchain)`. An unqualified label inherits the *current* toolchain, which inside a device `BUILD.gn` is the device toolchain.

"`gn desc` says no such target, but it's right there." `gn desc`/`path`/`refs` take a *fully-qualified* label including toolchain. To inspect a host instance, quote it: `gn desc out/default '//foo:bar(//build/toolchain:host_x64)'`.

"My header change didn't propagate to a consumer two levels up." You used `deps` where you needed `public_deps`. Public configs (include dirs) only flow transitively through `public_deps` chains (Fig. 3).

"A data file/helper binary is missing at runtime though it built fine." It was a build-time `deps` but is needed at runtime; it belongs in `data_deps` so the packaging metadata walk includes it.

"Editing a generated input didn't trigger a rebuild." The producing `action()` did not declare that input in `sources` (or omitted a `depfile`). Hermeticity violated; Ninja never saw the dependency.

"`gn gen` is slow / I touched a BUILD.gn and everything re-elaborated." That is expected: `gn gen` reruns on any `BUILD.gn` change. Keep gen-time work (`exec_script`) minimal; push real work into `action()` so Ninja, not GN, runs it.

"Why are there four copies of my object file?" Because four toolchains reference the target (device, host, shared, a variant). They are independent instances by design (Fig. 4); disk cost buys clean cross-compilation and instrumentation.

XIII. Tests in the Build

Tests are first-class build artifacts, and—predictably—they live across two toolchains. A *host* test is an ordinary `host_x64` executable run directly on your machine. A *device* test is a component packaged like any other and run on the target (or emulator), declared with `fuchsia_test_package()`. The `fx test` command builds the relevant targets and dispatches each to the right runner: host binaries execute locally; device tests are resolved as packages and launched under the Test Runner Framework, which provides each test a hermetic component realm.

The recurring toolchain gotcha appears here too. A test-collection `group` must reference host tests through the host toolchain explicitly, or GN will try to build them for the device:

Listing 20. Collecting host and device tests correctly.

```
group("tests") {
  testonly = true
  deps = [
    ":my_device_test",                  # built in the default (device) tc
    ":my_host_test($host_toolchain)",   # forced into the host tc
  ]
}
```

This single line—`($host_toolchain)` on a test target—is one of the most common things a newcomer omits, and the symptom ("my host test won't build / links against the wrong libc") is the canonical multi-toolchain confusion in miniature. The same SDK contract that gates production code also has a compatibility-test surface (CTF) that exercises each frozen API level, ensuring out-of-tree code built against level N keeps working.

XIV. Drivers in the Build

Drivers (the subject of the companion graphics reference) build through the same machinery with two wrinkles: a driver is a *shared library* loaded by a driver host rather than a standalone executable, and it carries *bind rules* that the driver framework matches against hardware nodes. Both are ordinary build artifacts.

A DFv2 driver is declared with `fuchsia_cc_driver()` (producing the `.so`) wrapped in a `fuchsia_driver_component()` (adding the manifest and the compiled bind bytecode), then placed in a driver package. The bind rules—a small DSL matching device properties such as PCI vendor/device IDs—are compiled by the `bind` compiler (a host tool, Section VIII) into bytecode the driver manager evaluates at runtime.

Listing 21. A DFv2 driver target (sketch).

```
import("//build/drivers.gni")

driver_bind_rules("bind") {
  rules = "my_gpu.bind"          # bind compiler -> bytecode
  bind_output = "my_gpu.bindbc"
}

fuchsia_cc_driver("driver") {     # the loadable .so
  output_name = "my_gpu"
  sources = [ "gpu_driver.cc" ]
  deps = [ "//sdk/fidl/fuchsia.gpu.magma:fuchsia.gpu.magma_cpp" ]
}
```

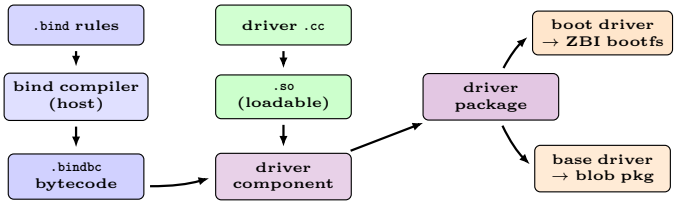

Fig. 10. Driver build path. The bind compiler (host) emits matchable bytecode; the driver `.so` and manifest form a driver component; boot-critical drivers land in the ZBI bootfs, the rest in blob packages loaded later.

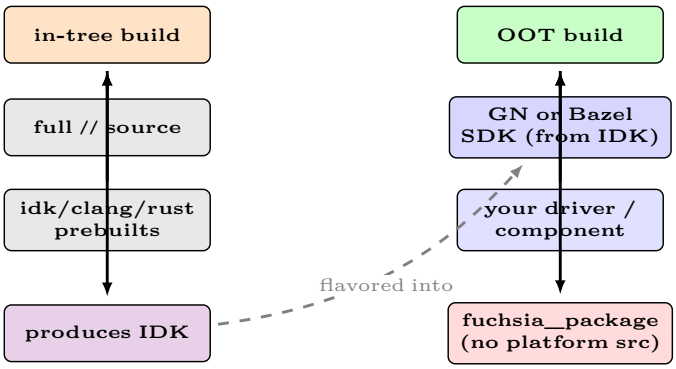

Fig. 11. In-tree vs. out-of-tree. The in-tree build consumes all source and prebuilts and *emits* the IDK; OOT builds consume an SDK flavored from that IDK and never see platform source.

```
fuchsia_driver_component("component") {
  manifest = "meta/my_gpu.cml"
  deps     = [ ":driver", ":bind" ]
  info     = "my_gpu-info.json"
}
```

The boot-vs-base split (Fig. 10) is the build-time face of the bootstrapping constraint from Section IX: a driver needed to bring up storage must be in the ZBI's bootfs (it cannot live in a package on a volume that does not yet exist), while everything else ships as a normal package resolved after boot. Assembly decides which is which, from the board configuration—another instance of *what is compiled* (the same driver `.so`) being decoupled from *where it is placed*.

XV. The SDK / IDK and Out-of-Tree Builds

Not everyone builds the whole platform. Driver and component authors frequently build *out-of-tree* (OOT) against an SDK rather than the full `//` source tree.

The raw deliverable is the **IDK** (Integrator Development Kit): the bare set of headers, prebuilt platform libraries, FIDL definitions, and tools that constitute Fuchsia's contract to external developers. The IDK is deliberately *build-system-agnostic*—it "does not contain any reference to toolchains or build systems" [10]. Concrete SDKs are produced *from* the IDK by adding a build integration: the **GN SDK** and the **Bazel SDK** are two such flavors.

OOT builds depend only on libraries the SDK exposes. In the Bazel SDK these arrive through the `@fuchsia_sdk` (and, for unstable driver surfaces, `@internal_sdk`) external repositories [11]. Internally the IDK is assembled from GN *SDK atoms* (one per exported library/tool) collected into *SDK molecules*; each atom contributes its headers, prebuilt, and a metadata record, and the whole set is described by an SDK manifest—so producing the IDK is itself a metadata walk (Section III) over the in-tree graph.

A. API Levels and Platform Versioning

Each SDK targets a defined set of **API levels**; an API level freezes a snapshot of the platform surface so that a component built against level N keeps working as the platform advances. FIDL drives this directly: declarations carry `@available` annotations naming the API level at which a method, type, or field was `added`, `deprecated`, or `removed`.

Listing 22. Versioned FIDL with @available.

```
@available(added=12)
closed protocol Echo {
    @available(added=12)
    strict EchoString(struct { value string:64; })
        -> (struct { response string:64; });
    @available(added=15, deprecated=20)
    strict EchoLegacy(struct { v string; }) -> ();
};
```

When `fidlc` compiles for a target API level, it includes exactly the elements available at that level; the resulting surface is recorded in `.api` (and golden-file) checks that fail the build if an incompatible change slips in. The package's ABI revision stamp (Section IX) then ties the built artifact to the level it was compiled against, and the device's resolver enforces compatibility at load time. The net effect: API leveling is a build-time *and* runtime contract, and it is implemented through the same code-generation and metadata machinery as everything else—there is no separate versioning subsystem, only `@available` flowing through `fidlc`.

XVI. The Bazel Migration

Fuchsia is incrementally adopting **Bazel** as its primary build system (RFC-0186), migrating one component at a time and bringing the Bazel SDK into `fuchsia.git` [11]. Two coexistence modes are in play: (a) the GN/Ninja build invokes Bazel for some subgraphs (a `bazel_action`-style bridge), and (b) OOT developers consume the Bazel SDK directly. Starlark rules—`fuchsia_package`, `fuchsia_component`, `fuchsia_cc_driver`—mirror the GN templates.

The hybrid bridge is worth understanding because it is how the migration proceeds without a flag day. A GN `bazel_action()` target wraps an invocation of `bazel build` on a chosen Starlark target, treating Bazel as just another action in the Ninja graph: GN hands Bazel a workspace (an SDK assembled in-tree plus the Bazel rules), Bazel builds its subgraph, and the outputs flow back as the action's declared `outputs`. To GN it is an opaque action with inputs and outputs; to Bazel it is an ordinary build. This lets a single component migrate to Bazel rules while the rest of the OS still builds under GN/Ninja—the two systems meet at a hermetic action boundary, exactly the seam that hermeticity (Section XVII) was designed to support.

Listing 23. A Bazel BUILD with Fuchsia rules (OOT).

```
load("@fuchsia_sdk//fuchsia:defs.bzl",
     "fuchsia_component", "fuchsia_package")

cc_binary(name = "bin", srcs = ["main.cc"],
          deps = ["@fuchsia_sdk//pkg/fdio"])

fuchsia_component(
    name = "component",
    manifest = "meta/my.cml",
    deps = [":bin"])

fuchsia_package(name = "pkg", components = [":component"])
```

The most instructive contrast for understanding both systems is how they pick toolchains. GN is *explicit*: you write the toolchain into the label, `:gen($host_toolchain)`, and the developer is responsible for naming the right context. Bazel is

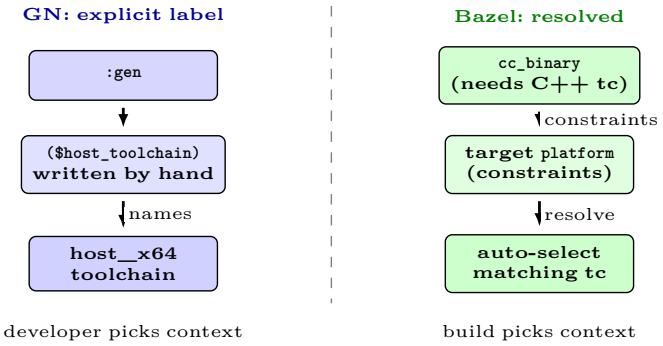

Fig. 12. Two answers to "which toolchain?". GN requires an explicit toolchain label in the dependency; Bazel infers it by matching target `platform` constraints against registered toolchains.

TABLE VIII
GN concept ↔ Bazel concept.

GN	Bazel
`BUILD.gn`	`BUILD.bazel`
`gn gen` (analysis)	analysis/loading phase
Ninja (execution)	Bazel execution phase
toolchain in label `(...)`	platform + constraint resolution
`source_set`/`static_library`	`cc_library`
`group`	`filegroup`
`action`	`genrule` / custom rule
`declare_args`	build settings / `-define`/flags
`config`	`copts`/`features`/`toolchain`

resolved: targets declare required `constraint_value`s, the build defines `platform`s, and Bazel's toolchain-resolution algorithm automatically selects a registered toolchain whose constraints satisfy the target platform [11]. The same cross-compilation problem—host tool vs. device binary—is solved by manual labeling in GN and by automatic constraint matching in Bazel (Fig. 12).

XVII. Build Performance & Caching

Two structural properties keep an OS-scale build tractable.

The gen/run split. Because `gn gen` is separate from `ninja`, a source edit that does not touch any `BUILD.gn` skips elaboration entirely and goes straight to an incremental Ninja run. Ninja's model—a static dependency graph plus file timestamps, with `restat` to prune no-op rebuilds when a regenerated file is byte-identical—means a one-line change recompiles only what truly depends on it.

Remote Build Execution (RBE). Compile and link actions can be dispatched to a remote worker pool via **reclient**, which wraps each action with a `rewrapper` prefix (controlled by GN args) that ships inputs to the service and retrieves outputs [12]. RBE provides both massive parallelism (hundreds of remote cores) and a shared action cache: an action whose inputs hash to a previously-seen value returns the cached output without recompiling. C++ RBE is enabled by default on `linux-x64` hosts. The `fx rbe` subcommands (`preflight`, `last_logdir`) manage credentials and logs.

Hermeticity is the enabling property. Remote execution and caching only work if an action's output is a pure function of its declared inputs. This is why GN/Ninja are so strict about `sources`, `outputs`, and `depfile` declarations (Section III): the input set must be *complete* so its hash is meaningful. An action that reads an undeclared file breaks two ways at once—it fails to rebuild locally when that file changes, *and* it returns stale or wrong results from the remote cache. The discipline that feels pedantic in a single local build is exactly what unlocks a 10× speedup across a fleet. For the hardware engineer the parallel is exact: a cacheable action is a combinational function of its inputs; a hidden input is undeclared state, and undeclared state defeats both memoization and reproducibility.

The two-phase win, restated. Because elaboration (`gn gen`) is separated from execution (`ninja`), and because Ninja's graph plus content hashing lets RBE memoize individual actions, the incremental cost of a one-line edit is bounded by the transitive closure of *that line*, not the size of the tree. A multi-million-file OS stays editable on a laptop because almost nothing actually rebuilds.

TABLE IX
Build introspection commands.

Command	Answers
`gn desc OUT //t:x`	resolved deps, configs, flags, toolchain of a target
`gn path OUT //a //b`	a dependency path from `//a` to `//b`
`gn refs OUT //t:x`	reverse deps: who depends on `//t:x`
`fx gn refs ...`	same, via the `fx` wrapper
`ninja -t graph`	dump the Ninja dependency graph
`ninja -t query F`	inputs/outputs of a single file/rule
`fx build -v`	verbose: show the actual commands run

XVIII. Debugging the Build

When the graph does something surprising—a target built in the wrong toolchain, an unexpected dependency, a missing file—GN's introspection commands answer the question directly (Table IX). The recurring trick is to remember the toolchain suffix: `gn desc` and friends operate on a *fully-qualified* label, so to inspect a host instance you must write `'//foo:bar(//build/toolchain:host_x64)'` (quoted, because the parentheses are shell metacharacters).

A worked example: "why is `//src/foo:bar` being built for the device when I only wanted the host test?" Run `gn path out/default '//:default' '//src/foo:bar'` to find the chain pulling it in, then `gn desc` on the offending intermediate to see whether the dependency edge omitted the `($host_toolchain)` suffix—the single most common cause of "it built for the wrong platform."

XIX. How the Build Got Here

The current shape is easier to hold in mind with its history, because each step removed a seam that newcomers used to trip over.

Two builds, then one. Early Fuchsia had *two* build systems: a Zircon build ("ZN") for the kernel and core, and a userspace GN build, glued by a generated bridge. Engineers had to understand both and the handoff between them. Unifying everything under one GN invocation—with the kernel relegated to a set of *toolchains* (Section VII) rather than a separate build—eliminated the single biggest source of build confusion. The multi-toolchain model was the enabling idea: once a toolchain

TABLE X
Quick glossary of build-system terms.

Term	Meaning
GN	"Generate Ninja"; the meta-build/elaboration engine.
Ninja	Executor of the flattened build graph.
jiri	Source + prebuilt manifest/sync tool.
CIPD	Pinned-binary package service for pre-builts.
fx / ffx	Workflow wrapper / device-facing tool.
toolchain	Named compiler+linker+flags+args context.
label	`//dir:name(toolchain)` target identifier.
variant	Toolchain plus instrumentation (asan, ...).
config	Reusable bundle of compiler/linker flags.
template	GN macro expanding into primitive targets.
metadata walk	Graph traversal collecting target metadata.
fidlc	FIDL frontend; `.fidl` → JSON IR (host).
fidlgen	FIDL backend; IR → language bindings (host).
cmc	Component manifest compiler; `.cml`→`.cm`.
blob	Content-addressed (Merkle-named) file.
meta.far	Package metadata archive; its hash names the package.
AIB	Assembly Input Bundle; a packaged platform piece.
ZBI	Zircon Boot Image: kernel + bootfs + bootargs.
blobfs/fxfs	On-disk blob filesystem; FVM is its volume manager.
IDK / SDK	Integrator Development Kit; build-flavored SDKs.
API level	Frozen platform surface snapshot (`.api` files).
RBE	Remote Build Execution (via reclient).

can be "freestanding, no libc, kernel mode," the kernel stops needing its own build system.

Hand-rolled images, then Product Assembly. Image construction was once baked into GN targets, entangling *what is compiled* with *what is shipped*. Splitting image construction into a standalone `ffx assembly` step (per RFC-0072) made assembly a pure-data recombination of pre-built artifacts, enabling one set of binaries to be assembled into many products and build types.

GN/Ninja, now incrementally Bazel. The move toward Bazel (RFC-0186) trades GN's explicit toolchain labels for constraint-based resolution and brings the SDK in-tree, but it preserves the two-phase analyze/execute shape and the hermetic-action contract. The migration rides on that contract: a Bazel subgraph is just a hermetic action inside the GN graph.

The throughline across all three changes is the same: *collapse special cases into the general multi-toolchain, hermetic-action model*. Understanding that model is therefore not just how to read today's build—it is the invariant the system keeps converging toward.

XX. Glossary

XXI. Conclusion

The Fuchsia build is best understood as an *elaboration* engine (GN) feeding a *build runner* (Ninja), with toolchains playing the role of cross-compilation cell libraries. The one idea that unlocks the whole system is that a target is not a binary—it is a *template for a binary*, instantiated once per toolchain context that references it. `//src/util:codec` is a design unit; `//src/util:codec(host_x64)` and `//src/util:codec(fuchsia:x64)` are two distinct instances mapped onto two distinct cell libraries, exactly as one Verilog module elaborates into two gate-level netlists against two process corners.

TABLE XI
One-line mental model per component.

Component	What it is
jiri	Lays out source + pinned CIPD prebuilts under `//`.
fx	Wrapper that records the build dir and drives gn/ninja.
GN	Elaboration: resolves graph, instantiates per toolchain, emits Ninja.
Ninja	Executor: runs compile/link incrementally.
Toolchain	Compiler+linker+flags context; a target builds once per toolchain.
Variant	A toolchain plus instrumentation (asan, coverage, ...).
fidlc/cmc	Host code generators; output compiled in the target lane.
Package	Content-addressed blob tree rooted at `meta.far`.
Assembly	Stitches packages + config into ZBI + filesystem image.
SDK/IDK	Exported platform contract for out-of-tree builds.

Everything else is consequence. Cross-compilation is just the target lane depending on the host lane via a `($host_toolchain)` suffix. FIDL is that pattern applied per interface: `fidlc` on the host, bindings compiled on the target. Variants are extra toolchains with instrumentation. Packages are content-addressed blob trees; assembly stitches them into a ZBI. The SDK is the in-tree build's exported contract, consumed OOT; the Bazel migration swaps explicit toolchain labels for constraint-based resolution but leaves the underlying two-phase shape intact.

For a hardware engineer, the payoff is conceptual economy: there is no magic in `fx build`, only an elaboration pass instantiating modules across libraries and a runner executing the result. Once the multi-toolchain instancing is internalized, the Fuchsia build stops being a black box and becomes what it actually is—a large, ordinary, and remarkably regular dependency graph.

References

[1] Google, "Fuchsia build system overview," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/build_system/fuchsia_build_system_overview

[2] Google, "Introduction to GN," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/build_system/intro

[3] Google, "GN toolchains and the Fuchsia Build," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/build_system/internals/toolchains/gn_toolchains_overview

[4] Google, "Build with GN labels," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/build_with_labels

[5] Google, "Build variants in the Fuchsia build," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/build/build_system/internals/toolchains/build_variants

[6] Google, "RFC-0097: FIDL Toolchain," *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0097_fidl_toolchain

[7] Google, “Fuchsia packages,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/packages/package
[8] Google, “Fuchsia Merkle Roots,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/security/merkleroot
[9] Google, “Software Assembly,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/software_assembly/overview
[10] Google, “RFC-0139: Bazel SDK,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0139_bazel_sdk
[11] Google, “RFC-0186: Bazel for Fuchsia,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0186_bazel_for_fuchsia
[12] Google, “Build with RBE,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.googlesource.com/fuchsia/+/main/build/rbe
[13] Google, “Go in Fuchsia,” *Fuchsia Documentation*, 2024. [Online]. Available: https://fuchsia.dev/fuchsia-src/development/languages/go