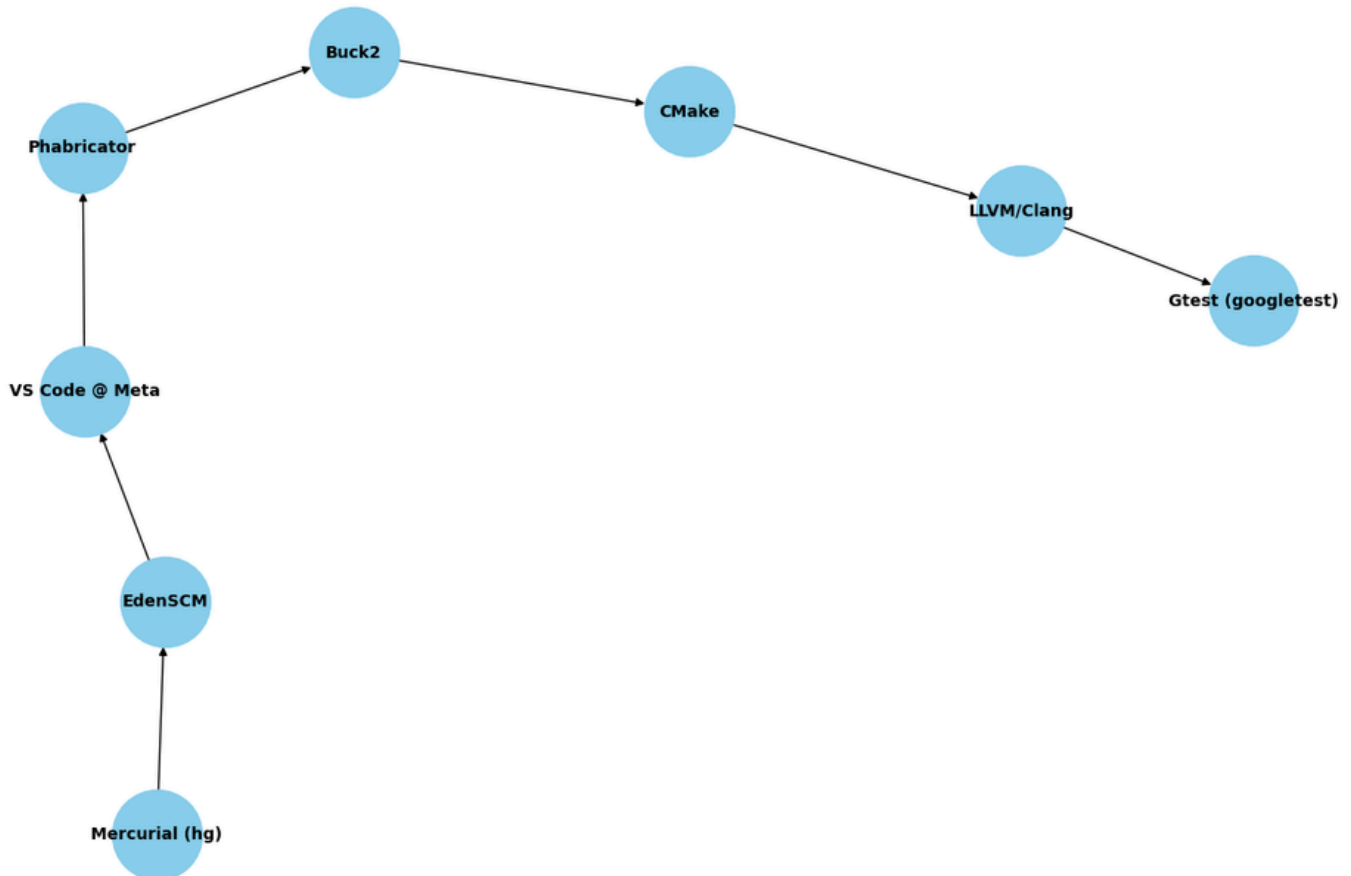


Meta Requests

Software development tools and environments:

1. **Mercurial (hg):** A distributed source control management system, similar to Git, but more scalable. It includes many custom internal extensions at Meta
2. **Buck2:** A large-scale, fast, reliable, and extensible build tool developed and used by Meta; it supports a variety of languages and is available as an open-source project
3. **VS Code @ Meta:** A unified developer environment for various programming languages, bundled with extensions that integrate with Meta infrastructure and workflows
4. **Phabricator:** A suite of web-based software development collaboration tools, including code review, repository browsing, and change monitoring
5. **EdenSCM:** A source control system that includes components like EdenFS and Mononoke, designed for scalability and performance with large repositories
6. **CMake:** An open-source, cross-platform build system generator used to build projects using platform- and compiler-independent configuration files
7. **LLVM/Clang:** A compiler for C, C++, and other languages, used for building and optimizing code
8. **Gtest (googletest):** A unit testing framework for C++

Workflow of Meta's Internal Development Tools



Flow Explanation:

1. **Mercurial (hg)** → Source control system used to manage code versions.
2. **EdenSCM** → Enhances Mercurial with scalable performance for large repositories.
3. **VS Code @ Meta** → Developer environment that interacts with EdenSCM for code editing.
4. **Phabricator** → Used for code review and collaboration, integrated with VS Code.
5. **Buck2** → Build system triggered post-review for compiling code.
6. **CMake** → Generates build configurations used by Buck2.
7. **LLVM/Clang** → Compiles the code using configurations from CMake.
8. **Gtest (googletest)** → Runs unit tests on the compiled code.

What is EdenSCM?

EdenSCM is a source control system—like Git or Mercurial—but designed for **very large codebases**. It was originally developed by Facebook (Meta) to manage their massive monorepos (repositories with millions of files and commits).

Why Use EdenSCM?

Most version control systems (like Git) slow down when:

- The repository has **millions of files**
- The **history is very long**
- Many developers are working at once

EdenSCM solves this by:

- Only downloading the files you actually use
- Using a **virtual filesystem** (EdenFS) to load files on demand
- Having a **server (Mononoke)** that sends just the needed data
- Offering a **fast CLI** (eden) that feels like Mercurial

Main Components

Component	What It Does
eden CLI	Command-line tool for developers to interact with the repo
Mononoke	Server that stores and serves repository data
EdenFS	Virtual filesystem that loads files only when needed

What's in the GitHub Repo?

 [bergwolf/eden GitHub Repo](#)

This is a **public fork** of the original EdenSCM project. It includes:

- Source code for the CLI, server, and filesystem
 - Build scripts (build.sh, build.bat)
 - CMake and Rust build files
 - Documentation and setup instructions
-

What is LLVM?

LLVM is like a **toolbox for building compilers**. A compiler is a program that turns human-readable code (like C++) into machine code that a computer can run.

- LLVM stands for **Low-Level Virtual Machine**, but today it's more of a **modular compiler framework**.
- It helps developers build tools that can **analyze**, **optimize**, and **compile** code for many different platforms (Windows, Linux, macOS, etc.).






What is Clang?

Clang is a **front-end** for LLVM. It understands programming languages like:

- C
- C++
- Objective-C
- Objective-C++

Clang reads your code, checks it for errors, and turns it into an intermediate form (LLVM IR), which LLVM then turns into machine code.

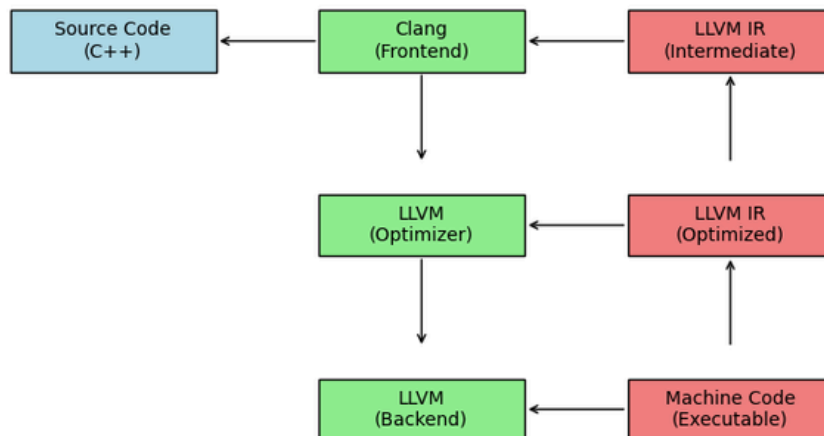
Why Use LLVM/Clang?

Feature	Why It's Useful
 Fast and Lightweight	Clang compiles code quickly and uses less memory than older compilers like GCC.
 Clear Error Messages	Clang gives very readable and helpful error messages.
 Modular Design	You can use just the parts you need—great for building tools like IDEs, linters, or static analyzers.
 Cross-Platform	Works on many operating systems and supports many CPU types.
 Great for Research	LLVM is used in universities and companies to build new programming languages and tools.

Real-World Uses

- **Apple** uses Clang/LLVM in Xcode.
- **Google, Mozilla, and Intel** use LLVM for performance-critical software.
- It's also used in **game engines, embedded systems, and machine learning compilers** (like TensorFlow XLA).

Here's a simple diagram that shows how **LLVM** and **Clang** work together to compile C++ code:



How It Works – Step by Step

- 1. Source Code (C++)**
You write your code in C++ (e.g., main.cpp).
- 2. Clang (Frontend)**
Clang reads your code, checks for errors, and converts it into an intermediate form called **LLVM IR (Intermediate Representation)**.
- 3. LLVM IR (Unoptimized)**
This is a low-level, platform-independent version of your code. Think of it as a simplified version of your program.
- 4. LLVM Optimizer**
This part improves your code by removing unnecessary steps, simplifying logic, and making it faster.
- 5. LLVM IR (Optimized)**
Now your code is cleaner and more efficient, but still in IR form.
- 6. LLVM Backend**
This converts the optimized IR into **machine code** specific to your computer's CPU.
- 7. Machine Code (Executable)**
Finally, you get an executable file (like a.out or .exe) that your computer can run.

Here's a **complete overview of LLVM and Clang commands**, categorized for clarity. These are based on the official documentation from the LLVM and Clang projects

Clang Commands (Frontend)

Clang is used to compile C, C++, Objective-C, and more.

◆ Basic Compilation

◆ Common Flags

- -Wall, -Wextra – Enable warnings
- -O0, -O1, -O2, -O3, -Os, -Ofast – Optimization levels
- -g – Include debug info
- -std=c++17 – Specify C++ standard
- -I<dir> – Add include path
- -L<dir> – Add library path
- -l<lib> – Link with library

◆ Emit LLVM IR

◆ Static Analysis

◆ Preprocessor

LLVM Toolchain Commands

These tools work with LLVM IR and bitcode.

◆ IR Tools

- llvm-as – Assemble LLVM IR to bitcode
- llvm-dis – Disassemble bitcode to LLVM IR
- llvm-link – Link multiple bitcode files
- opt – Optimize LLVM IR
- llc – Compile LLVM IR to assembly
- lli – Run LLVM IR directly (interpreter)

◆ Debugging & Analysis

- llvm-nm – List symbols in bitcode
- llvm-objdump – Disassemble object files
- llvm-dwarfdump – Dump DWARF debug info
- llvm-cov – Code coverage
- llvm-profdata – Profile data tool

◆ Developer Tools

- bugpoint – Reduce test cases
- FileCheck – Pattern matching for test output

- tblgen – Generate C++ code from table descriptions



Testing & Diagnostics

- lit – LLVM Integrated Tester
- diagtool – Explore Clang diagnostics
- llvm-exegesis – Benchmark machine instructions

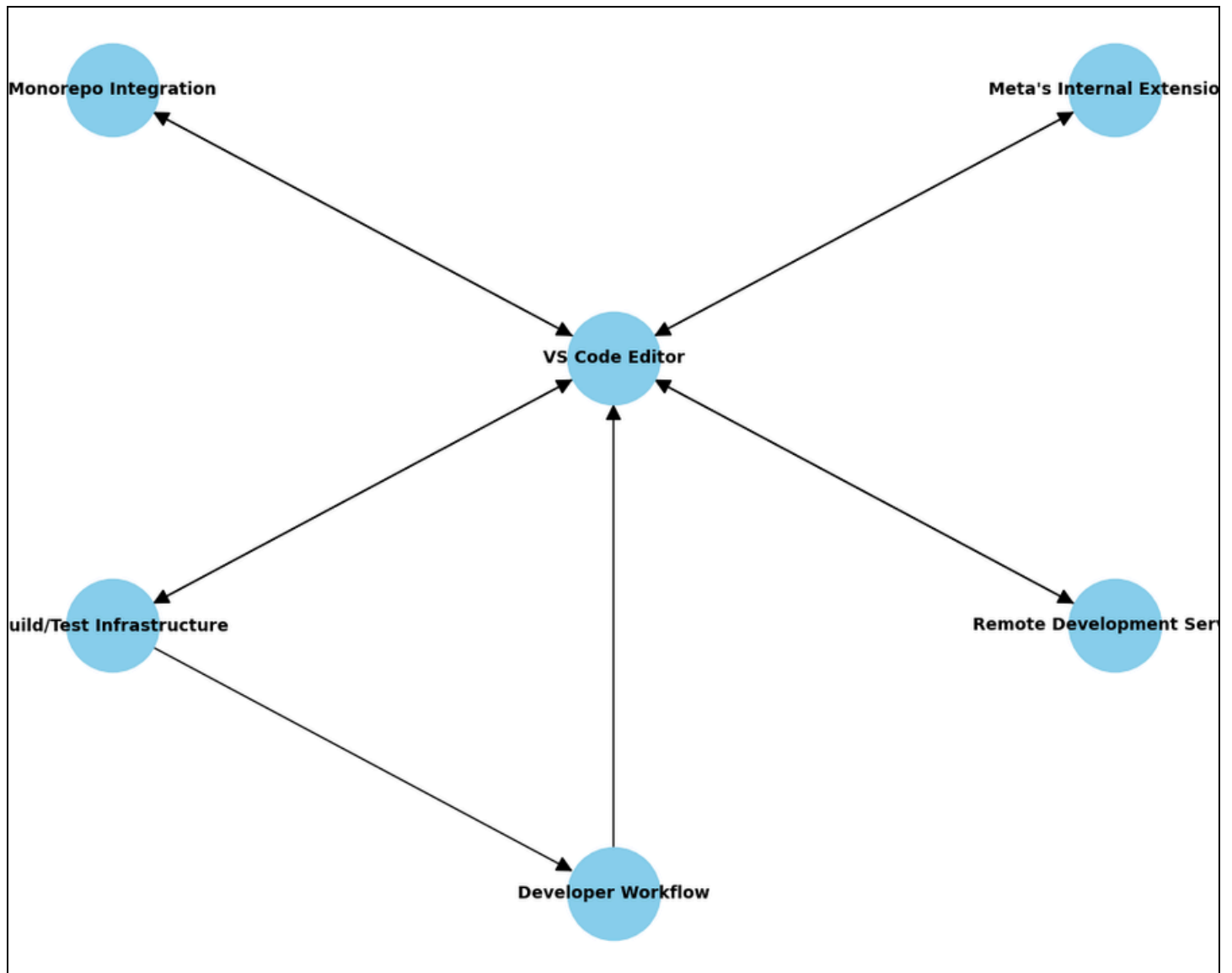


GNU Binutils Replacements

LLVM provides drop-in replacements for many GNU tools:

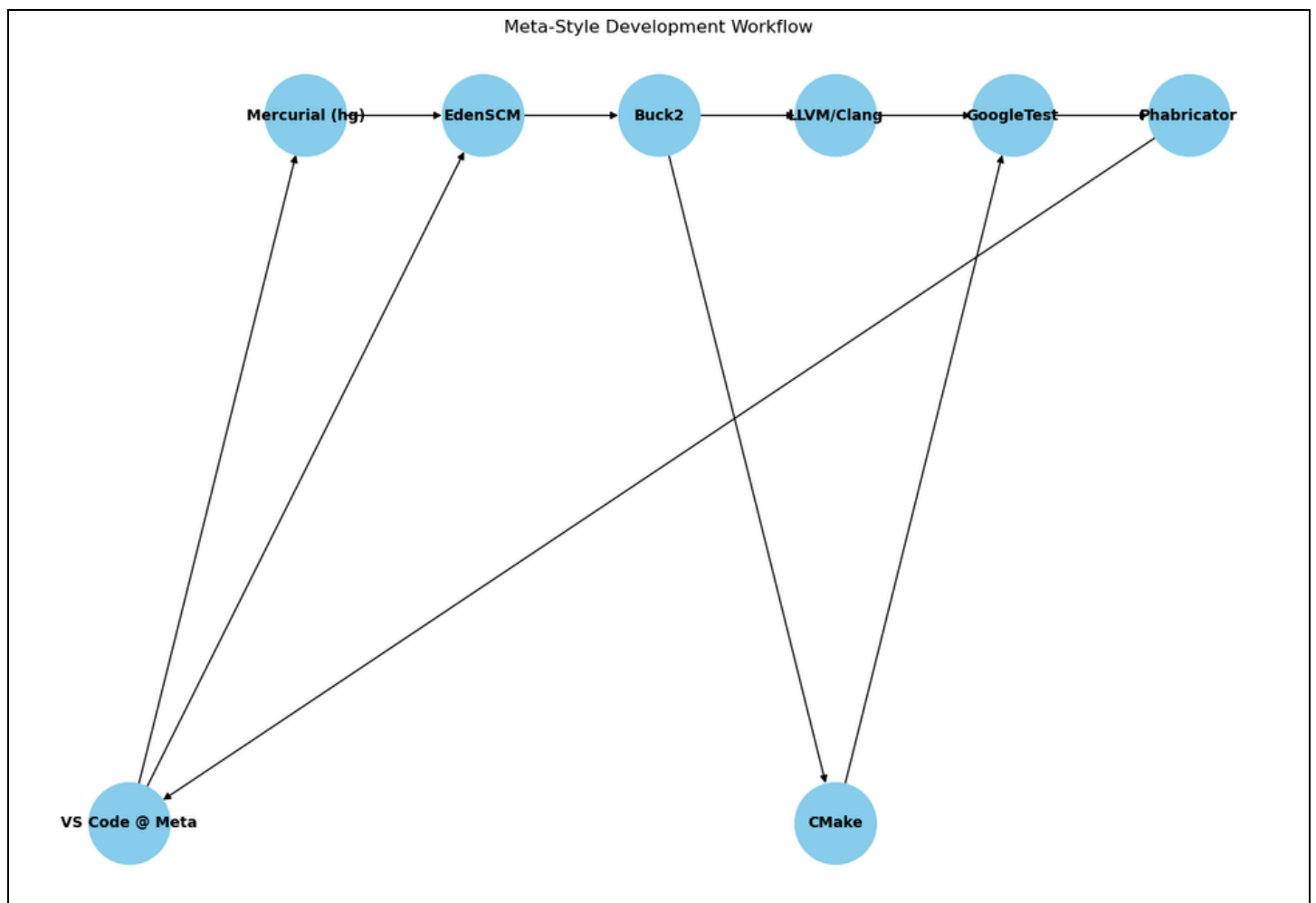
- llvm-ar, llvm-ranlib, llvm-objcopy, llvm-strip, llvm-size, llvm-readobj, llvm-addr2line, etc.
-

- VS Code Editor is the central tool used by developers.
- It connects to:
 - Meta's Internal Extensions for custom tooling.
 - Remote Development Servers for powerful compute.
 - Monorepo Integration to handle massive codebases.
 - Build/Test Infrastructure for CI/CD and validation



- VS Code Editor is the central tool used by developers.
- It connects to:
 - Meta's Internal Extensions for custom tooling.
 - Remote Development Servers for powerful compute.
 - Monorepo Integration to handle massive codebases.
 - Build/Test Infrastructure for CI/CD and validation.
- All components feed back into the **Developer Workflow**, creating a fast, integrated loop.

Here is a comprehensive workflow diagram showing how various tools interact in a Meta-style development environment:



Workflow Breakdown

1. **VS Code @ Meta**
 - The central development environment where engineers write and edit code.
 - Integrates with version control tools and internal extensions.
2. **Mercurial (hg)**
 - Used for version control.
 - Developers commit and manage code history here.
3. **EdenSCM**
 - Acts as a virtual filesystem and performance layer over Mercurial.
 - Efficiently handles large monorepos by loading files on demand.
4. **Buck2**
 - Meta's build system.
 - Takes code from EdenSCM and builds it using rules defined in BUCK files.
5. **LLVM/Clang**
 - Compiles C/C++ code during the build process.
 - Integrated into Buck2 for performance and optimization.
6. **CMake**
 - Used in projects that require cross-platform build configuration.
 - Can be invoked by Buck2 or used independently.
7. **GoogleTest**
 - Runs unit tests on the compiled code.

- Integrated into the build pipeline for continuous testing.

8. Phabricator

- Used for code review and collaboration.
 - Receives test results and diffs for review before merging.
-

Here's a complete list of what you can test using GoogleTest (GTest) on the IOTG (Internet of Things Group) platform, especially in embedded or systems-level development:

✓ What You Can Test with GoogleTest on IOTG

1. Application Logic

- Business rules
- State machines
- Data processing algorithms
- Protocol parsers (e.g., MQTT, CoAP)

2. Hardware Abstraction Layer (HAL)

- GPIO, I2C, SPI, UART interfaces (via mocks/stubs)
- Sensor drivers (mocked)
- Peripheral control logic

3. Middleware Components

- RTOS task scheduling logic
- Inter-process communication (IPC)
- Event queues and timers

4. System Services

- Bootloader logic
- Power management routines
- Configuration and calibration modules

5. File Systems and Storage

- FAT/exFAT/Flash file system logic
- EEPROM/Flash read-write routines (mocked)

6. Communication Stacks

- Network protocol handlers (TCP/IP, BLE, Zigbee)
- Message encoding/decoding
- Error handling and retries

7. Security Modules

- Authentication and encryption logic
- Secure boot validation
- Key management (mocked)

8. Error Handling and Logging

- Fault detection and recovery
- Logging and diagnostics
- Watchdog and reset logic

Types of Tests You Can Run

Test Type	Description
Unit Tests	Test individual functions or classes in isolation
Mocked Tests	Replace hardware calls with mocks to simulate behavior
Integration Tests	Combine modules and test their interaction
Regression Tests	Ensure new changes don't break existing functionality
Boundary/Edge Tests	Test limits of input ranges and error conditions

Tools Often Used Alongside GTest in IOTG

- **GoogleMock:** For mocking hardware interfaces
- **PlatformIO:** For running tests on embedded targets
- **CMake:** For building test suites
- **QEMU or Simulators:** For running tests without real hardware

What Is Clang?

Clang is a **compiler**—a tool that turns your C, C++, or Objective-C code into something your computer can run (machine code). It's part of the **LLVM project**, which is a collection of tools for building and optimizing software.

What Is the “Getting Started” Guide?

The Clang Getting Started Guide helps you:

- **Download and install Clang**

- **Build Clang from source** (if you want to customize it)
- **Understand how to use Clang** to compile your code
- **Set up your development environment**

It's like a beginner's manual for using Clang effectively.

? Why Is This Needed?

If you're a developer working with C or C++ (especially in systems, embedded, or performance-critical software), you need a **reliable, fast, and modern compiler**. Clang is:

- **Faster** and uses less memory than older compilers like GCC
- **Easier to understand** because of its clear error messages
- **Highly customizable** if you want to build your own tools or compilers

🔧 What Can You Do With Clang?

Here's what Clang helps you with:

Use Case	What It Means
✅ Compile Code	Turn .cpp or .c files into programs
🔍 Analyze Code	Find bugs or performance issues
🛠️ Build Tools	Create your own compilers or static analyzers
🚀 Optimize Code	Make your programs run faster
🧪 Test Embedded Systems	Use Clang with platforms like IOTG for cross-compilation