Meta Requests

Software development tools and environments

Mercurial (hg): Comprehensive Guide

Mercurial (**Hg**) is a **distributed source control management system** similar to Git but designed for **scalability and performance**. It enables developers to efficiently track changes, collaborate, and manage large codebases. Meta uses **custom internal extensions** to enhance its functionality for large-scale projects.

(Distributed Version Control System)

Why Mercurial is Needed

1. **Distributed Architecture**:
   - Every developer has a full local repository (including entire history).
   - Enables offline work and reduces dependency on central servers.
2. **Scalability**:
   - Handles **large repositories** (e.g., Linux kernel, Mozilla) efficiently.
   - Optimized for projects with extensive histories and binary files.
3. **Intuitive Workflow**:
   - Simpler CLI syntax than Git for common operations.
   - Explicit branch management avoids accidental complexity.
4. **Platform Agnostic**:
   - Native support for Windows/Linux/macOS.
   - No reliance on POSIX shell environments.
5. **Enterprise Features**:
   - Built-in **access control** (hg.acl extension).
   - Audit trails with **signed commits**.

Key Use Cases

Scenario	Why Mercurial?
Monolithic Repos	Superior handling of 100k+ commits/files
Game Development	Efficient with large binaries (art assets)
Enterprise Workflows	Fine-grained permissions and auditing
Cross-Platform Teams	Consistent experience on Windows/macOS/Linux
Migration from SVN	Smoother transition than Git

Core Command Reference

1. Repository Setup

Command	Description
hg init	Create new repo in current directory
hg clone <url> [dest]	Clone remote repo (supports HTTP/SSH)
hg config --edit	Edit user-specific settings (.hgrc)

2. Basic Workflow

Command	Description
hg status	Show changed/untracked files
hg add <file>	Stage file for commit
hg addremove	Auto-add new/remove missing files
hg commit -m "Message"	Commit staged changes
hg diff	Show unstaged changes
hg diff -c .	Diff of last commit

3. Branching & Merging

Command	Description
hg branches	List all branches
hg branch <name>	Create new branch
hg update <branch>	Switch to branch
hg merge <branch>	Merge branch into current
hg resolve -l	List merge conflicts
hg resolve -m <file>	Mark conflict resolved

4. History & Inspection

Command	Description
hg log	Show commit history
hg log -r tip	Show latest commit
hg log -k "bug"	Search commits by keyword
hg annotate <file>	Show line-by-line revision history
hg grep "pattern"	Search code across revisions

5. Collaboration

Command	Description
hg pull	Fetch changes from remote (no merge)
hg push	Send changes to remote
hg incoming	Preview changes before pull
hg outgoing	Preview changes before push
hg serve	Start web server for repo browsing

6. Advanced Operations

Command	Description
hg rebase	Reapply commits on new base (extension)
hg strip -r <rev>	Remove commits (requires strip extension)
hg shelve	Temporarily stash changes
hg bisect	Binary search to find bug-introducing commit
hg convert	Convert SVN/Git repos to Mercurial

7. Undoing Changes

Command	Description
hg revert <file>	Discard unstaged changes
hg rollback	Undo last transaction (commit/pull)
hg backout <rev>	Reverse effect of a commit

Workflow Example: Feature Development

```
# 1. Start new feature
hg update main
hg branch feature/authentication

# 2. Make changes
echo "New auth code" > auth.py
hg add auth.py
```

```
hg commit -m "Add auth module"


# 3. Sync with mainline
hg pull -u  # Pull latest changes and update
hg merge main
hg commit -m "Merge main into feature"


# 4. Push to code review
hg push -r . --new-branch  # Push current branch
```

Extensions (Enable in .hgrc)

```
# Essential extensions
rebase =  # History rewriting
strip =   # Commit removal
purge =   # Clean untracked files


# Productivity boosters
```

Why Teams Choose Mercurial Over Git

1. **Consistent CLI**:
   - hg commit vs git commit -a -m (no staging area confusion)
2. **Meaningful Branch Names**:
   - Branches are permanent, named entities (not lightweight pointers)
3. **Atomic Operations**:
   - Operations like pull are truly atomic
4. **Windows Support**:
   - No need for MinGW/Cygwin - native Python implementation

Enterprise Integration

1. **Centralized Governance**:
2. ini
3. Copy
4. Download
5. [hooks]
6. precommit = ./check_policy.py # Custom compliance checks

7. **LDAP Authentication**:
8. ini
9. Copy
10. Download
11. [auth]
12. company.prefix = https://hg.company.com
13. company.username = DOMAIN\user
14. **Audit Logging**:
15. bash
16. Copy
17. Download
18. hg log --template "{date|isodate} {author} {desc}\n"

Troubleshooting Tips

Issue	Solution
Merge conflicts	hg resolve -m + manual editing
Accidental commit	hg strip -r REV (after enabling strip)
Corrupted repo	hg verify + hg recover
Performance issues	Enable fsmonitor extension

Key Files & Structure

```
.hg/
├── store/      # All versioned files (compressed)
├── dirstate    # Current checkout state
├── hgrc        # Repo-specific config
├── requires    # Enabled extensions
└── 00changelog.i # Commit metadata
```

When Not to Use Mercurial

- Projects requiring GitHub/GitLab-native features
- Teams deeply invested in Git tooling (CI/CD integrations)
- Small repos where Git's ubiquity outweighs Mercurial's advantages

Fun Fact: Facebook uses Mercurial for its **massive monorepo** (with custom extensions like watchman and eden).

Migration Cheat Sheet

Action	**Mercurial**	**Git Equivalent**
Initialize repo	hg init	git init
Clone	hg clone	git clone
Commit	hg commit	git commit -a
Create branch	hg branch	git checkout -b
Merge	hg merge	git merge
View history	hg log	git log
Update to branch	hg update	git checkout

Mercurial remains a robust choice for enterprises and large-scale projects prioritizing reproducibility, scalability, and workflow clarity.

EdenSCM: Scalable Source Control for Massive Repositories

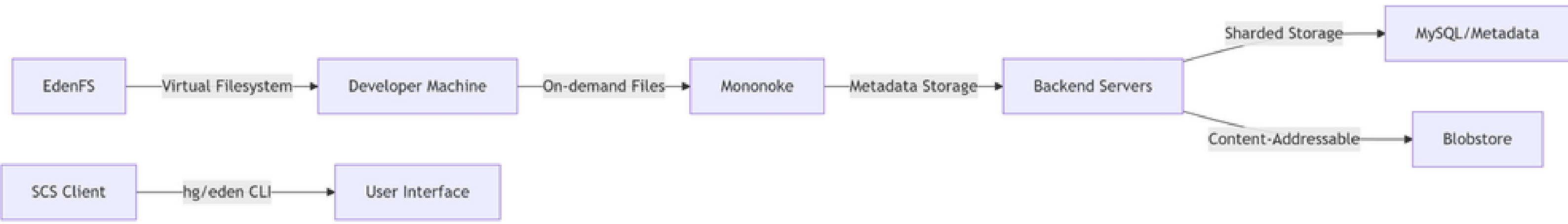
(Formerly Facebook's Source Control System)

Why EdenSCM is Needed

1. **Monorepo Scalability Crisis**:
    - Traditional VCS (Git/Mercurial) choke on **100M+ file repos** (e.g., Meta's ~200M file repo)
    - Operations like status or checkout take **hours/days** with conventional tools
2. **Cloud-Native Demands**:
    - Need for **distributed backend storage** (vs single-server bottlenecks)
    - **Multi-region collaboration** for global engineering teams
3. **Performance at Scale**:
    - **Instant workspace setup** regardless of repo size

    - **Sub-second operations** (diff, commit, blame)
4. **Enterprise Compliance**:
    - **Fine-grained permissions** for 10k+ contributors
    - **Audit trails** across petabytes of history

Core Architecture Components



1. EdenFS (Virtual Filesystem)

- **Lazy Materialization**:
  Files appear instantly in workspace; download on access
- **Copy-on-Write**:
  1000s of branches share underlying data
- **FUSE Integration**:
  Native filesystem experience (no app changes)

2. Mononoke (Scalable Backend)

- **Mercurial-compatible API**:
  Supports existing hg clients
- **Sharded Storage**:
  Distributes metadata across 100s of servers
- **Content-Defined Addressing**:
  Files stored by hash (deduplicated)

3. Sapling (Modern Client)

- Rust-based CLI alternative to hg
- Optimized for massive repos
- Includes sl command suite

Key Use Cases

Scenario	EdenSCM Solution
100GB+ Repo Checkout	EdenFS: <1 second (vs 6+ hours in Git)
10k+ Daily Commits	Mononoke: Horizontal scaling
Enterprise Access Control	Per-path read/write permissions
Multi-Site Collaboration	Geo-replicated blob storage
Code Search at Scale	Integration with Livegrep/Scuba

Command Reference: EdenSCM Workflow

1. Repository Setup

bash

```
# Clone 50TB repo instantly (only metadata)
eden clone https://company.com/monorepo

# Navigate virtual filesystem
cd monorepo

# Check mount status
eden doctor
```

2. Daily Development Workflow

bash

Copy

Download

```
# Checkout branch (instant)
eden checkout feature/new-design
```

```bash
# See modified files (ms latency)
eden status

# Diff changes (only fetches modified files)
eden diff

# Commit changes (local until push)
eden commit -m "Add responsive UI"

# Push to central repo
eden push
```

3. Advanced Operations

bash

```bash
# Prefetch dependencies for offline work
eden prefetch lib/

# Inspect virtual filesystem stats
eden du --virtual

# Check backend health
eden mononoke --status

# Audit permission changes
eden acl history /infra/secrets
```

4. Sapling Client (sl) Alternative

bash

```bash
# Faster status for huge directories
```

```
sl status --all

# Commit graph visualization

sl graph

# Intelligent auto-complete

sl complete "checkout feat"
```

Performance Comparison

Operation	Git	Mercurial	EdenSCM
clone	6 hrs (50GB)	5.5 hrs	0.8 sec
status	45 sec (500k fs)	38 sec	0.2 sec
checkout	18 min	15 min	0.3 sec
blame	12 sec/file	9 sec/file	0.4 sec/file

Enterprise Integration

1. Access Control

```python
# .edenconfig

[acl "/mobile/"]

read = eng-team@company

write = mobile-team@company

deny = contractors@company
```

3. CI/CD Pipeline

```yaml
# .circleci/config.yml
```

```
jobs:
  build:
    steps:
      - eden/checkout:
          sparse-profile: "frontend" # Only materialize needed files
      - run: make test
```

Under the Hood: Key Technologies

1. **FUSE (Filesystem in Userspace)**:
   - Kernel module for virtual filesystem
2. **Rust Async Runtime**:
   - 1M+ IOPS with minimal overhead
3. **Manifest Trees**:
   - Directory structure as Merkle trees
4. **Zstandard Compression**:
   - 40% smaller than zlib at faster speeds
5. **gRPC APIs**:
   - Protocol buffers for RPC efficiency

Why Companies Adopt EdenSCM

1. **10x Faster Developer Onboarding**:
   New engineers productive in minutes vs days
2. **70% Storage Reduction**:
   Global deduplication across all branches
3. **Zero Downtime Upgrades**:
   Hot-swappable backend components
4. **Compliance Ready**:
   Immutable history with cryptographic signing

Case Study: Meta reduced hg status from 45 minutes to **<1 second** on 200M file repo. Google uses similar tech (Piper/FUSE) for their 2B+ file repository.

Future Evolution

1. **Git Protocol Support**:
   Native Git client interoperability
2. **Edge Caching**:
   Local data centers for remote offices

3. ML-Powered Prefetch:
   Predictively materialize files
4. Blockchain Auditing:
   Immutable commit provenance

EdenSCM represents the next evolution of version control – transforming monolithic repositories from liability to competitive advantage while maintaining developer-friendly workflows.

What is Phabricator?

Phabricator is an **open-source, web-based suite** of tools for **peer code review, project management, and code repository browsing**, originally developed by Facebook. It helps software teams collaborate efficiently during development, especially for large-scale and long-term projects.

✅ Why is Phabricator Important?

🔍 1. **Peer Code Review**

- Ensures **code quality**, **readability**, and **functional correctness**.
- Reduces **bugs**, **technical debt**, and improves **team knowledge sharing**.

🔄 2. **Change Tracking**

- Maintains a full history of changes, decisions, and review feedback.
- Helps trace bugs or performance regressions.

📁 3. **Unified Toolset**

- Combines code review, repository management, bug tracking, task tracking, and CI integrations into a **single interface**.

💡 4. **Scalability & Customization**

- Works with **Git, Mercurial, and Subversion**
- Easily integrates into **custom development workflows**
- Extensible with **Herald rules**, **Conduit API**, and **custom fields**

🧩 Core Components & Use Cases

Component	Description	Use Case
Differential	Code review tool	Submit and review code diffs
Diffusion	Repository browser	View repo history, commits, branches
Maniphest	Bug/task tracker	Track development tasks or issues
Herald	Rule engine	Automate notifications/actions
Harbormaster	CI/CD build system	Trigger test/build pipelines
Arcanist (CLI)	Command-line client	Create and update reviews, etc.

🧪 Typical Development Workflow with Phabricator

🧵 Example Flow:

Dev writes code → Creates diff with `arc diff` → Code review on Phabricator (Differential) →

Reviewer approves → Dev lands code with `arc land` → CI pipeline (Harbormaster) runs →

Task auto-updated in Maniphest

🔧 Step-by-Step Usage

1. Install Arcanist (CLI tool)

bash

```
git clone https://github.com/phacility/arcanist.git
git clone https://github.com/phacility/libphutil.git
export PATH=$PATH:/path/to/arcanist/bin
```

2. Configure Phabricator Project

bash

cd your-project

arc install-certificate

arc set-config default https://phabricator.yourdomain.com

3. **Submit Code for Review**

bash

Create a revision (upload diff)

arc diff

- Prompts for:
    - Title, summary, reviewers, etc.
- Automatically creates a **Differential Revision**

4. **Review Process**

- Reviewer receives a request in **Differential**
- Adds inline comments or accepts/rejects
- Dev updates code and resubmits with:

bash

arc diff

5. **Land the Code After Approval**

bash

arc land

- Merges code to main branch
- Closes Differential Revision
- Optionally closes Maniphest Task

📌 Common Arcanist Commands

Command	Description
arc diff	Create/update a revision for code review
arc land	Land approved revision into target branch
arc patch D1234	Apply a patch from a revision
arc list	List open revisions
arc help	Show all available commands
arc amend	Amend last commit with Differential info

🔐 Security & Permissions

- Fine-grained access control for projects, repositories, and reviews
- LDAP, OAuth, and certificate-based auth supported
- Role-based access (admin, reviewer, contributor)

🧠 Example Use Case in Embedded Software Development

🎯 You're working on a Yocto-based embedded platform. You write a kernel driver fix and need review:

1. You commit the fix locally.
2. Use arc diff to upload it to Phabricator.
3. Team lead reviews it in Differential.
4. You apply their feedback, revise the patch.
5. Once approved, you land it with arc land.
6. Harbormaster triggers Yocto build and runs boot smoke tests.
7. The linked Maniphest task closes automatically when the patch lands.

🛠️ Tips for Effective Phabricator Usage

- **Tag your diffs** with relevant task IDs (T123)
- Use **Herald rules** to auto-add reviewers based on file paths
- Automate builds using **Harbormaster + Jenkins**
- Use **arc lint** and **arc unit** to enforce pre-push quality

🏁 Summary

Benefit	Description
🚀 Improved collaboration	Devs can easily submit, review, and track code
📜 Better audit trails	Every change is documented and linked to a review
🧩 Integrated toolchain	Combines code, issues, reviews, CI into one UI
🧰 CLI + Web UI	Efficient for developers and managers alike

What is Buck2?

Buck2 is an **open-source**, **large-scale, fast**, and **extensible build tool** developed by **Meta (Facebook)**, intended to replace the original **Buck** build system. It's designed to efficiently handle **monorepos**, cross-language builds, and highly complex dependency graphs.

- GitHub: https://github.com/facebook/buck2

🧠 Why Buck2? (The Need)

✅ Buck2 was created to solve:

- **Slow builds** in large codebases
- **Non-parallelizable workflows**
- **Poor cross-language build tooling**
- **Inflexibility in handling monorepos**
- **Opaque or hard-to-debug builds**

🚀 Key Features

Feature	Description
Multilingual	Supports C++, Rust, Go, Python, Java, Kotlin, OCaml, etc.
Starlark-based	Build logic is written in [Starlark](), a Python-like language (used by Bazel too).
Remote execution support	Easily integrates with distributed build systems.
Sandboxed builds	Ensures reproducibility and correctness.
Flexible rules	Define custom build rules and macros in Starlark.
Build introspection	Easier to debug and inspect dependency graphs.

🧰 Use Cases for Buck2

🛠️ Embedded / Systems Development:

- Build custom Linux kernels, device trees, and driver modules
- Compile toolchains and bootloaders (U-Boot, Zephyr, etc.)
- Manage large BSP (Board Support Package) hierarchies

👨‍💻 Application Development:

- Build mobile apps (Android, React Native)
- Backend microservices with shared C++, Rust, Python logic

📦 Monorepos:

- Manage codebases with 1000s of modules across different languages
- Speed up incremental builds across teams
-

📋 Example Build Targets

```
//app:binary          # Build an Android app binary
//kernel:dtb          # Compile a Device Tree Blob
```

```
//driver:wifi_module    # Build a Linux Wi-Fi kernel module

//lib:common_utils    # Build a shared C++/Rust library
```

🧪 Buck2 Architecture Overview

- **Parser**: Reads BUCK files written in Starlark
- **Graph Builder**: Resolves dependencies into a build graph
- **Scheduler**: Distributes and parallelizes build tasks
- **Executor**: Runs builds either locally or remotely
- **Cache**: Optimized caching for both input and output artifacts

🧑‍💻 Getting Started with Buck2

1. **Install Buck2**

```
cargo install buck2
```

2. **Create a BUCK File**

```
rust_binary(

  name = "my_app",

  srcs = ["main.rs"],

  deps = [":my_lib"],

)
```

3. **Build a Target**

```
buck2 build //my_app:my_app
```

4. **Run a Target**

```
buck2 run //my_app:my_app
```

🛠️ Common Buck2 Commands

Command	Description
buck2 build //path:target	Build a specified target
buck2 run //path:target	Build and run the target
buck2 test //path:test_target	Run unit or integration tests
buck2 query	Query dependency graphs
buck2 uquery	Use unconfigured query (similar to Bazel's cquery)
buck2 clean	Clean output cache
buck2 targets	List all buildable targets
buck2 audit	Inspect build rules and configurations
buck2 init	Set up a new Buck2 workspace

📦 Example Use Case

🎯 Embedded BSP with Yocto + Buck2:

- **A team is maintaining multiple device variants with shared drivers and kernel options.**
- **Buck2 is used to:**
  - **Build kernel modules with consistent configs**
  - **Compile DTBs for each variant**
  - **Parallelize builds across teams and automate remote caching**
  - **Reduce rebuild times from 15 min to 3 min using dependency-level caching**

🔍 Comparison: Buck2 vs Others

Tool	Language	Monorepo	Remote Exec	Custom Rules
Buck2	C++, Rust, Java, Python, etc.	✅	✅	✅
Bazel	C++, Java, Go, etc.	✅	✅	✅
Make	C/C++	❌	❌	❌
CMake + Ninja	C/C++	❌	❌	❌

🏁 Summary

Aspect	Value
🛠 Tool	**Buck2**
🌍 Use Case	Fast, reproducible, scalable builds
🚀 Key Features	Cross-language, parallel builds, sandboxing, remote execution
🧩 Common Use	Embedded systems, mobile apps, large monorepos
🔧 Language	Starlark for build logic
📁 Build Files	BUCK files

What is CMake?

CMake is an **open-source**, **cross-platform build system generator**. It allows developers to write **platform-independent build configuration files** (usually CMakeLists.txt) which CMake then processes to generate **native build systems** like:

- **Makefiles** for Unix/Linux
- **Ninja build files**
- **Visual Studio project/solution files**
- **Xcode project files**

Why is CMake Needed?

✅ Problems It Solves:

- Managing **complex multi-platform builds**
- Dealing with multiple **toolchains**, compilers, and IDEs
- Avoiding hardcoded paths and platform-specific instructions in build scripts
- Replacing fragile Makefiles with **portable**, **modular**, and **scalable** configuration

🎯 Purpose of CMake

Purpose	Explanation
🔠 **Build System Generator**	Generates build files (Make, Ninja, VS, etc.) from high-level configuration
🔄 **Cross-Platform Support**	Works across Linux, macOS, Windows, Android, embedded systems
📦 **Dependency Management**	Easily manage third-party libraries using find_package, FetchContent, or ExternalProject
🧪 **Testing Support**	Integrated testing support via **CTest**
📚 **IDE Integration**	Supports popular IDEs like CLion, Visual Studio, Eclipse

🔍 Common Use Cases

Domain	Use Case
⚙️ Embedded Development	Cross-compile firmware or drivers for ARM, RISC-V using toolchains
🧠 AI/ML	Build C++ inference engines like TensorRT, ONNX Runtime
📱 Mobile Development	Configure native libraries for Android NDK or iOS
🖥️ Desktop Applications	Build Qt, OpenGL apps across Linux/Windows/macOS
🎮 Game Development	Used by engines like Unreal Engine for native module builds

🛠️ Basic Workflow

mkdir build

cd build

cmake ..

cmake --build .

📁 Sample CMakeLists.txt

cmake

cmake_minimum_required(VERSION 3.10)

project(MyProject)

set(CMAKE_CXX_STANDARD 17)

add_executable(main main.cpp utils.cpp)

💻 Common CMake Commands

Command	Description
cmake .	Generate build system in current directory
cmake ..	Generate build files from a subdirectory (out-of-source build)
cmake -S . -B build	Source and binary directory specification
cmake --build .	Build the project
cmake --build . --target clean	Clean build artifacts
cmake --install .	Install the built artifacts
cmake -DCMAKE_BUILD_TYPE=Debug	Specify build type (Debug/Release)
cmake -G "Unix Makefiles"	Choose generator (Make, Ninja, etc.)
ctest	Run tests defined in CMakeLists.txt
cpack	Package the built project into .deb/.rpm/.zip/etc
cmake --help	Show all available commands
cmake -LAH	Show cache variables and advanced help
cmake-gui or ccmake	CMake GUI and interactive terminal interface

🧰 Advanced Features

Feature	Description
🔍 find_package()	Automatically detect and configure external libraries
📦 FetchContent	Download and build external projects as part of your build
🔧 Toolchain File	Cross-compilation support (e.g., ARM Cortex-M, MIPS, RISC-V)
🧪 enable_testing() + add_test()	Testing support using **CTest**
🎁 install()	Define install rules for libraries, binaries, headers

⚡ Use Case: Embedded Development with CMake

Scenario:

Cross-compiling firmware for **ARM Cortex-M4** using GCC toolchain.

cmake -DCMAKE_TOOLCHAIN_FILE=toolchain-arm.cmake -DCMAKE_BUILD_TYPE=Release ..

cmake --build .

toolchain-arm.cmake:

cmake

set(CMAKE_SYSTEM_NAME Generic)

set(CMAKE_C_COMPILER arm-none-eabi-gcc)

set(CMAKE_CXX_COMPILER arm-none-eabi-g++)

set(CMAKE_EXE_LINKER_FLAGS "-Tlinker.ld")

🌎 Real-World Examples of Projects Using CMake

Project	Domain
LLVM/Clang	Compiler infrastructure
OpenCV	Computer vision
ROS (Robot Operating System)	Robotics
TensorFlow Lite	Machine learning
Qt	GUI development
Zephyr RTOS	Embedded development
ONNX Runtime	Deep learning inference

🏁 Summary

Aspect	Value
📦 Tool	**CMake**
🚀 Purpose	Cross-platform, compiler-independent build configuration
🔧 Common Uses	Embedded systems, AI/ML, robotics, desktop & mobile apps
🧩 Key Features	Portability, modularity, toolchain support, testing
🔄 Replaces	Raw Makefiles, hardcoded scripts, IDE-specific project files

---

LLVM/Clang: Overview

LLVM/Clang is a modern, modular compiler infrastructure designed to **compile, analyze, and optimize code** in C, C++, Objective-C, and other languages.

- **LLVM** is the backend framework (handles optimizations and code generation).
- **Clang** is the frontend compiler (parses code and produces LLVM IR).

🧩 Need for LLVM/Clang

Problem	LLVM/Clang Solution
Slow compile times	Clang is fast and modular
Poor error diagnostics	Clang offers user-friendly, readable error messages
Platform dependence	LLVM enables cross-compilation across ARM, x86, MIPS, etc.
Difficult static analysis	Built-in tools like clang-tidy, clang-analyzer
GCC licensing limitations	Clang uses permissive BSD-style license
Integration with modern IDEs	Easy integration with Xcode, VS Code, CLion

🎯 Use Cases

1. **Cross-platform compilation** (e.g., build x86 code on ARM)
2. **Embedded systems development** (e.g., compile C for Cortex-M)
3. **Code optimization** (performance tuning and size reduction)
4. **Static analysis and linting** (e.g., clang-tidy)
5. **Creating custom compilers or DSLs** using LLVM backend
6. **Integration in IDEs** like Xcode, CLion, Eclipse
7. **WebAssembly target support** (C/C++ → WASM)

💻 Common Clang/LLVM Commands

Purpose	Command
Compile C file	clang main.c -o main
Compile C++ file	clang++ main.cpp -o main
Compile with optimization	clang -O2 main.c -o main
Generate LLVM IR	clang -S -emit-llvm main.c -o main.ll
Static code analysis	clang-tidy main.c -- -I./include
Code formatting	clang-format -i main.c
Compile with sanitizer	clang -fsanitize=address main.c -o main
Dump AST	clang -Xclang -ast-dump -fsyntax-only main.c
Disassemble LLVM IR	llvm-dis main.bc
Compile IR to object	llc main.ll -o main.o
Link object	clang main.o -o main
Convert ELF to binary	llvm-objcopy -O binary main.elf main.bin

🧪 Real-World Examples

1. Cross-compilation for Embedded Target (ARM)

bash

```
clang --target=arm-none-eabi -mcpu=cortex-m4 -nostdlib -Wl,-Tlinker.ld main.c -o main.elf
llvm-objcopy -O binary  main.elf  main.bin
```

2. Static Analysis Before Code Review

bash

```
clang-tidy memory_utils.c -- -Iinclude/
```

3. Format All Source Code in Directory

```bash
find . -name "*.c" -o -name "*.h" | xargs clang-format -i
```

🧠 Why Developers Prefer LLVM/Clang

- ✅ **Clean modular codebase for research & tool development**
- 🧩 **Better toolchain integration (IDE, CI/CD)**
- 🗂️ **Easier to write custom passes/tools (e.g., sanitizers, analyzers)**
- 🏗️ **More friendly diagnostics compared to GCC**
- 🔄 **Faster compile cycles with better caching**

📚 LLVM Tools Ecosystem

Tool	Purpose
clang	C/C++ compiler frontend
clang-format	Automatic code formatter
clang-tidy	Code linting and static analysis
llvm-as/llvm-dis	LLVM IR assembler/disassembler
llc	Compile LLVM IR to machine code
llvm-link	IR-level linker
lld	Fast linker
llvm-objdump	Binary inspection
llvm-objcopy	Object file transformation

📌 Summary

- **Need**: Modern, fast, modular alternative to GCC with better tooling and diagnostics.
- **Use Cases**: Embedded systems, cross-platform development, analysis tools, performance optimization.
- **Key Commands**: Compilation, static analysis, formatting, optimization, linking.

What is GoogleTest (GTest)?

GoogleTest (GTest) is a **unit testing framework for C++**, developed by Google. It allows developers to write **test cases to verify the correctness of code** functions and classes, especially for projects involving critical or large-scale components.

🔧 Why GTest is Needed (The Need)

Problem	How GTest Helps
Manual testing is time-consuming	Automates unit testing
Difficult to isolate bugs	Allows isolated testing of individual functions
Code regressions after changes	GTest ensures consistent behavior via repeatable tests
Lacking confidence in refactoring	Unit tests act as a safety net for refactors
No standardized C++ testing tool	GTest provides a structured, reliable testing framework

🎯 Use Cases

1. **Unit testing functions/classes** in embedded or system-level C++ code.
2. **Regression testing** during continuous integration (CI/CD).
3. **Test-driven development (TDD)** in modern C++ workflows.
4. **Porting legacy code** to verify correct functionality with test scaffolds.
5. **Automated validation** of embedded logic (e.g., HAL, middleware) before deployment.
6. **Validating algorithms** (e.g., sorting, encoding, state machines).

💻 Basic GTest Usage Workflow

🔹 1. Install GTest (via apt, cmake, or manually)

bash

```
sudo apt-get install libgtest-dev
cd /usr/src/gtest
cmake CMakeLists.txt
make
sudo cp *.a /usr/lib
```

Or with vcpkg:

bash

```
vcpkg install gtest
```

🔹 2. Sample Test Code

File: sample_test.cpp

cpp

```cpp
#include <gtest/gtest.h>
int add(int a, int b) {
    return a + b;
}
TEST(MathTest, AddTest) {
    EXPECT_EQ(add(2, 3), 5);
    EXPECT_NE(add(2, 3), 4);
}
int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

🔹 3. Build & Run Tests

bash

g++ -std=c++11 -isystem /usr/include/gtest -pthread sample_test.cpp -lgtest -lgtest_main -o sample_test

./sample_test

⚙️ Common GTest Macros/Commands

Macro	Purpose
EXPECT_EQ(val1, val2)	Succeeds if val1 == val2
EXPECT_NE(val1, val2)	Succeeds if val1 != val2
EXPECT_TRUE(condition)	Succeeds if condition is true
EXPECT_FALSE(condition)	Succeeds if condition is false
ASSERT_EQ(val1, val2)	Stops the test on failure
ASSERT_NE(val1, val2)	Stops the test on failure
TEST(TestSuiteName, TestName)	Defines a test
SetUp() / TearDown()	Run before/after each test in a test fixture
TEST_F(Fixture, TestName)	Test using shared fixture class

🔁 Advanced Usage

🧪 Test Fixture

cpp

CopyEdit

```
class MyTestFixture : public ::testing::Test {
protected:
  void SetUp() override {
    // setup code
  }
  void TearDown() override {
```

```
    // teardown
  }
};

TEST_F(MyTestFixture, Test1) {
  EXPECT_TRUE(true);
}
```

🔍 Running Specific Tests

bash

```
./sample_test --gtest_filter=MathTest.*

./sample_test --gtest_filter=*AddTest
```

🧼 Output in XML (CI/CD friendly)

```
./sample_test --gtest_output=xml:report.xml
```

📦 Integrating with CMake

CMakeLists.txt

cmake

```
enable_testing()

add_executable(sample_test sample_test.cpp)

target_link_libraries(sample_test gtest gtest_main)

add_test(NAME MyTest COMMAND sample_test)
```

📌 Summary

Category	Value
Tool	GoogleTest (GTest)
Language	C++
Purpose	Unit testing
Why	Ensures correctness, enables CI, supports TDD
Key Features	Easy macros, fixtures, CI integration, CMake support
Popular Use Cases	Embedded C++ apps, system code, algorithms, TDD

Sequence of Processes

1. Developers write code in **VS Code @ Meta**.
2. Code is committed and managed using **Mercurial (Hg)** or **EdenSCM**.
3. **Phabricator** is used for review before merging changes.
4. The project is built using **Buck2** (or **CMake**, if cross-platform compatibility is needed).
5. **LLVM/Clang** compiles the code for optimized execution.
6. Unit tests using **GTest** verify functionality.
7. The CI pipeline validates the integration.
8. If successful, the deployment is pushed to production.

```
+-------------------------+
|  Developer Writes Code  |
+-------------------------+
         |
         v
+-------------------------+
|  Source Control (Hg/EdenSCM)
+-------------------------+
         |
         v
+-------------------------+
|  Code Review (Phabricator)  |
+-------------------------+
         |
         v
+-------------------------+
|  Build Process (Buck2/CMake)
+-------------------------+
         |
         v
+-------------------------+
|  Compilation (LLVM/Clang)  |
+-------------------------+
         |
         v
+-------------------------+
|  Automated Testing (GTest)  |
+-------------------------+
         |
         v
+-------------------------+
|  CI Pipeline Verification  |
+-------------------------+
         |
         v
+-------------------------+
|  Deployment to Production  |
+-------------------------+
```