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.

Mercurial (hg) is a **distributed version control system (DVCS)** written in **Python** and optimized for **performance and scalability**.

- It's **similar to Git**, but with an emphasis on **simplicity, usability**, and **strong command-line interface consistency**.
- The command hg is used to interact with Mercurial (hg = symbol for mercury).

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

Q1: What's the difference between Git and Mercurial?

A: Both are DVCS tools. Git offers more flexibility and is widely adopted, but Mercurial provides a simpler and more consistent CLI. Mercurial emphasizes user-friendliness and safety.

Q2: How does branching in Mercurial differ from Git?

A: Mercurial supports:

- **Named branches** (persist in history)
- **Anonymous branches** (short-term development)
- **Bookmarks** (similar to Git branches — lightweight, movable)

Q3: What is the .hg folder?

A: It's the **metadata folder** that contains the entire history of the repository, configuration, and revision logs — critical for all Mercurial operations.

Why Mercurial?

- **Design Goals**:
  - High performance (handle Linux-scale projects).
  - Simple, consistent command-line interface.
  - Fully distributed architecture.
  - Cross-platform (Python-based).
- **Advantages**:
  - No server dependency (all clones are full repositories).
  - Secure SHA-1 hashing for data integrity.
  - Efficient branching/merging model.

1. **Distributed VCS** (Git/Mercurial) solved critical flaws in centralized systems:
   - Single point of failure.
   - Lack of offline access.
   - Inefficient branching.
2. **Mercurial** emerged as a user-friendly, high-performance DVCS alternative to Git.
3. Modern VCS prioritizes **speed**, **decentralization**, and **scalability**.

Core Concepts and Definitions

Concept	Description
Repository (repo)	A collection of tracked files and their revision history
Changeset	A snapshot of the project at a given time (a commit in Git)
Working Directory	Local copy of project files that can be modified
.hg Directory	Metadata directory storing history, settings, branches, etc.
Branch	A named line of development
Tag	A human-readable label for a specific changeset
Revision ID	A unique identifier (hash) for each changeset
Clone	A complete copy of a repository, including its full history
Pull / Push	Sync changes between repositories
Merge	Combine changes from different lines of development

Key Notes:

- No explicit **staging area** like Git; changes go directly into a changeset.
- .hg/ folder is central to all operations (like .git/ in Git).

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

- Repository (Repo) : A directory storing all project files and their complete history.
- Created via:
- `hg init        # Creates a new repo in the current directory`
- `hg clone <path> # Clones an existing repo (local or remote)`

Working Directory

- Your local filesystem view of the repo.
- Changes here are *not* saved until committed.

Changeset (Commit)

- A snapshot of the project at a point in time.
- Identified by a **unique 40-digit hex ID** (e.g., b56ce7b07c52) or a local **revision number** (e.g., 0, 1).
- Created via:
-
- `hg commit -m "Descriptive message"`

2. Basic Workflow

[Track Changes]

```
hg add <file>      # Start tracking a new file
hg remove <file>   # Stop tracking a file (deletes it)
hg status          # Show untracked/modified/deleted files
hg diff            # View changes in working directory
```

[Commit Changes]

```
hg commit -m "Fix login bug"  # Commit staged changes
```

View History

```
hg log          # Full commit history
hg log -l 3       # Last 3 commits
hg log -v         # Detailed commit info (files changed)
```

3. Collaboration & Sharing

Clone a Repo

```
hg clone https://hg.example.com/project  # Clone remote repo
```

Pull Updates

```
hg pull        # Fetch changes from remote
hg update        # Apply pulled changes to working dir
```

```
# Or combine:
```

```
hg pull --update
```

Push Changes

```
hg push        # Send local commits to remote repo
```

Check Sync Status

```
hg incoming      # Show changes not pulled
hg outgoing      # Show changes not pushed
```

4. Advanced Operations

Update Working Directory

```
hg update <rev>    # Switch to a specific revision
```

```
hg update tip     # Switch to latest commit
```

Branching & Merging

- Create Branch:
-
- `hg branch new-feature  # Start a new branch`

- Merge Branches:
- `hg update main       # Switch to target branch`
- `hg merge new-feature   # Merge branch`
- `hg commit -m "Merge"   # Commit the merge`

Resolve Conflicts

- If files conflict during merge/update:
  1. Edit conflicted files (look for <<<<<<<, =======, >>>>>>> markers).
  2. Mark as resolved:
  3.
  4. **hg resolve --mark <file>**
  5. Commit the resolved state.

5. Tagging & Ignoring Files

Create a Tag

hg tag v1.0 # Mark current revision as "v1.0"

hg push --tags # Push tags to remote

Ignore Files

- Create .hgignore in repo root:
-
- ***syntax: glob # Use simple patterns***
- ****.log # Ignore all .log files***
- ***build/ # Ignore build directory***

6. Key Features & Best Practices

- **Atomic Commits**: Every commit is a complete, immutable snapshot.
- **Distributed Model**: Every clone is a full backup with complete history.
- **Lightweight Branches**: Branching is metadata-only (no extra directories).
- **Performance**: Optimized for large repos (e.g., handling 100k+ files).
- **Extensions**: Enhance functionality (e.g., purge, color, rebase).

7. Essential Commands Cheatsheet

Command	Purpose
hg init	Create new repo
hg clone <url>	Clone remote repo
hg status (hg st)	Show changed/untracked files
hg diff	Show changes
hg commit (hg ci)	Commit changes
hg push	Send commits to remote
hg pull	Fetch remote changes
hg update (hg up)	Switch revisions/branches
hg merge	Merge branches
hg log	View history
hg help <command>	Get command-specific help

8. Pro Tips

- **Revision Identifiers**: Use short IDs (e.g., b56ce7b07c52 → b56ce7b).
- **Undo Mistakes**:
-
- `hg revert <file>    # Discard uncommitted changes`
- `hg rollback        # Undo last commit (use cautiously!)`
- **Web Interface**:
- `hg serve           # Start web server for repo browsing`

Why Merging is Necessary

- **Divergent Histories**: When two developers modify the same file(s) in separate clones/branches.
- **Branch Collaboration**: Integrating changes from feature branches into a main branch.

- **Update Conflicts**: Occurs when pulling remote changes that conflict with local uncommitted work.

2. Merge Workflow Step-by-Step

Scenario:

- You cloned a repo (hg clone), made local commits, and need to integrate upstream changes.

Steps:

- `hg pull          # Fetch changes from remote (does NOT modify working directory)`
- `hg heads         # View all branch heads (shows divergence)`
- `hg merge         # Merge remote changes into local working directory`
- `hg commit -m "Merge" # Commit the merge result`
- `hg push          # Share merged changes`

3. How Mercurial Handles Merges

Automatic Merges (Safe Cases):

- **Different Files**: Changes to unrelated files.
- **Same File, Different Regions**: Non-overlapping edits within a file.
- **Identical Changes**: Same edit made independently.

Manual Merge Required (Conflicts):

- **Overlapping Edits**: Changes to the same line(s) in a file.
- **Structural Conflicts**: File renamed in one branch, modified in another.

4. Conflict Resolution

Identify Conflicts:

```
hg merge        # Attempt automatic merge

# Output:

#   merging file.txt

#   warning: conflicts while merging file.txt! (edit, then use 'hg resolve --mark')
```

Conflict Markers in Files:

```
plaintext

<<<<<<< local (your changes)

Your version of the code
```

```
=======

Incoming version of the code

>>>>>>> other (remote changes)
```

Resolve Conflicts:

1. Edit conflicted file(s) to remove markers and choose desired code.
2. Mark as resolved:
3. `hg resolve -m file.txt # Mark file.txt resolved`
4. Commit the merge.

5. Merge Tools

Configure external tools for visual conflict resolution (e.g., kdiff3, meld, vscode):

```
# Edit ~/.hgrc

[ui]

merge = kdiff3

# Usage during merge:

hg merge --tool=kdiff3
```

Common Tools:

- **kdiff3: Cross-platform diff/merge**
- **meld: GNOME-based visual tool**
- **vimdiff: Terminal-based resolution**

6. Advanced Merge Scenarios

Merging Unrelated Branches:

```
hg merge --force    # Force merge of unrelated histories (rarely needed)
```

Undoing a Bad Merge:

```
hg update -C <pre-merge-revision>  # Abandon merge

hg backout <merge-commit>        # Reverse merge via new commit
```

File-Level Operations:

```
hg resolve --list    # List conflicted files (U = unresolved)
```

```
hg resolve --all    # Attempt auto-resolve on all files

hg resolve --unmark file.txt  # Revert to conflicted state
```

7. Best Practices for Smooth Merges

1. **Pull Frequently**: Reduces divergence and conflict complexity.
2. **Commit Before Merging**: Never merge with uncommitted changes (hg status should be clean).
3. **Merge in Small Chunks**: Smaller merges are easier to resolve.
4. **Test After Merging**: Validate functionality before pushing.
5. **Use Named Branches**: Explicit branches improve traceability.

8. Key Commands Cheatsheet

Command	Purpose
hg merge [branch]	Merge another branch into current
hg resolve -m <file>	Mark file as resolved
hg resolve -l	List merge conflicts
hg heads	Show all divergent heads
hg update -C	Abandon merge (clean update)
hg backout	Reverse a commit

9. How Mercurial Tracks Merges

- **Merge Commits**: Have two parent changesets.
- **DAG Structure**: Merges appear as convergence points in the revision graph.
- **Ancestry Tracking**: Mercurial knows which changes are already merged via revision history.

Visualize Merges:

```
hg log -G  # ASCII graph view

# Output:
```

```
# o   commit3 (merge) [parents: commit1, commit2]
# |\
# | o commit2 (remote)
# o | commit1 (local)
# |/
# o   base_commit
```

10. Pro Tips

- **Dry Run**: Preview merge effects without modifying working directory:
-
- `hg merge --preview`
- **Merge Drivers**: Customize merge logic per file type via [merge-patterns] in .hgrc.
- **Avoid "Commit Races"**: Pull immediately before committing to minimize conflicts.

1. Repository Structure (.hg Directory)

The .hg directory contains all metadata and history. Key components:

File/Directory	Purpose
store/	Core data storage (revlogs, manifests)
dirstate	Tracks working directory status (modified/added/removed files)
hgrc	Local configuration settings
bookmarks	Stores named pointers to commits (similar to Git branches)
tags	Version-controlled tag definitions
undo.*	Transaction rollback data

2. Revlog: The Storage Engine

Core Design

- **Append-only** structure for data integrity

- **Delta compression**: Stores changes relative to previous versions
- **Immutable**: Once written, revlogs never change

Revlog Types

1. **Changelog (00changelog.i/.d)**
   - Stores commit metadata (author, timestamp, parents, commit message)
   - Each entry points to a manifest node
2. **Manifest (00manifest.i/.d)**
   - Directory tree snapshot for each commit
   - Maps filenames to filelog nodes
3. **Filelogs (data/*.i/.d)**
   - Stores actual file content for each revision
   - One revlog per file

3. Data Model: Key Concepts

Node IDs

- **40-byte SHA-1 hashes** (e.g., b56ce7b07c52...)
- Calculated from:
- python
- ```python
hash = sha1(parent1 + parent2 + manifest_id + file_contents + metadata)
```

### Revision Addressing

| Identifier | Description | Example |
|---|---|---|
| Revision number | Local integer (not transferable) | 42 |
| Short node ID | Unique prefix (min 12 chars) | b56ce7b07c52 |
| Full node ID | Complete 40-char hash | b56ce7b... |
| Special symbols | tip (latest), . (current), null (empty) | hg update tip |

## 4. Delta Compression Mechanics

### Storage Optimization

- **Delta chains**: Store differences from previous revisions
- **Snapshot every N revisions**: Full text storage to limit chain length
```

- **Heuristic selection**: Choose optimal base revision for minimal delta size

Example Delta Chain

```
Rev 0: "Hello World"      (Full text)
Rev 1: Delta(0): "Hello Mercurial" → +"Mercurial", -"World"
Rev 2: Delta(1): "Hi Mercurial" → +"Hi", -"Hello"
```

5. Transaction System

Ensures atomic operations:

1. Write changes to temporary files
2. Update journal files (undo.*)
3. Atomically rename files on success
4. Rollback via journal if interrupted

```
hg commit
# Behind scenes:
#  1. Write new changelog entry to temp file
#  2. Update manifest revlog
#  3. Write filelogs
#  4. Atomically rename files → commit visible
```

6. Working Directory Management

Dirstate File

Binary format tracking:

- File modification times
- Size and mode flags
- Parent revisions
- Merge state

Update Process

```
hg update feature-branch
# 1. Compare manifest of target revision vs. dirstate
# 2. For changed files:
#   - Preserve local changes (if safe)
#   - Overwrite with target version (if clean)
```

```
# 3. Update dirstate metadata
```

7. Networking Protocol

SSH/HTTP Communication

1. Client sends command: lookup, changegroup, pushkey
2. Server computes deltas between known nodes
3. Transfers compressed bundles (.hg patches)

Bundle Format

```text
# V2 bundle header
HG20\x00\x00\x00\x00...
[Compressed changelog group]
[Manifest group]
[Filelog groups]
```

8. Performance Optimizations

- **File system caching**: Memory-map revlog indexes
- **Lazy loading**: Only read manifest/filelogs when needed
- **Copy tracing**: Detect renames via content similarity (no explicit tracking)
- **Revlog slicing**: Efficient retrieval of partial history

9. Integrity Guarantees

- **Hash verification**: All data referenced by SHA-1 hashes
- **Cross-revlog checks**: Manifest entries must match filelog existence
- **Append-only writes**: Prevents data corruption
- **Fsync options**: Configurable durability (tradeoff: performance vs. safety)

10. Extension Hooks

Mercurial exposes internal APIs for extensions:

```python
# Example: Pre-commit hook
def check_style(ui, repo, **kwargs):
    if "TODO" in open("file.txt").read():
        ui.warn("Commit rejected! Remove TODO first.\n")
```

```
    return 1
ui.setconfig("hooks", "precommit", check_style)
```

Key Takeaways

1. **Revlogs are immutable, compressed, and delta-based**
2. **All data is content-addressed via SHA-1**
3. **Transactions ensure atomic operations**
4. **Working directory state is tracked via dirstate**
5. **Distributed nature comes from bundle transfers**

1. Core Workflow Components

A. File States & Tracking

State	Symbol	Meaning	Commands
Untracked	?	Not in version control	hg add
Modified	M	Changed since last commit	hg commit
Added	A	Staged for next commit	hg forget to unstage
Removed	R	Scheduled for deletion	hg remove / hg restore
Clean	C	Unchanged since last commit	(Baseline state)

- **Key Insight**: Mercurial tracks *content*, not files. Renames are detected via content similarity.

2. Essential Daily Commands

A. Change Management

1. **Review Changes**:
   - hg status → Lists file states (short form: hg st)
   - hg diff → Shows line-level changes (use -w to ignore whitespace)
   - hg diff --stat → Summary of changed files
   -
2. **Commit Workflow**:

3. hg add <file>   # Stage new files
4. hg remove <file> # Stage deletions
5. hg commit   # Create changeset (opens editor)
   - **Pro Tip**: hg commit -A commits *all* changes (adds/removes/modifications).
6. Undoing Mistakes:
   - **hg revert <file> → Discard uncommitted changes**
   - **hg revert -a → Revert *all* working directory changes**
   - **hg commit --amend → Fix last commit's message/content**

B. History Navigation

Command	Purpose
hg log	Full commit history
hg log -l 5	Last 5 commits
hg log -k keyword	Search commits by keyword
hg log -r tip	Show latest commit
hg annotate <file>	See who changed each line (blame)
hg cat -r 3 <file>	Output file from revision 3

- Revision Identifiers:
  - tip: Latest commit
  - #0: First commit
  - 12ab: Unique hash prefix

C. Branching & Merging

1. Lightweight Branching:
2.
3. hg branch feature-x   # Create branch
4. hg commit -m "Start feature"
5. hg update main   # Switch back to main
6.
7. Merging:
8. hg update main
9. hg merge feature-x   # Merge feature into main
10. hg commit -m "Merge feature-x"
11.

12. **Conflict Resolution**:
    - Mercurial inserts conflict markers (<<<<<<<, =======, >>>>>>>)
    - Use hg resolve --list to see conflicts
    - After editing: hg resolve --mark <file>

3. Collaboration Tools

A. Sharing Changes

```
hg pull        # Fetch remote changes
hg update      # Apply to working directory
hg push        # Send local commits upstream
```

- Critical Flags:
  hg pull -u → Pull + update in one step
  hg push --force → Overwrite history (use sparingly!)

B. Remote Comparisons

- **hg incoming → Preview changes before pulling**
- **hg outgoing → See what will be pushed**

4. Advanced Daily Operations

A. Stashing Unfinished Work

```
hg shelve      # Temporarily store changes
hg unshelve    # Restore changes
```

B. Tagging Releases

```
hg tag v1.0    # Create permanent tag
hg tags        # List all tags
```

C. Configuration Tweaks

Edit ~/.hgrc for:

```ini
[ui]
username = Your Name <email@example.com>
editor = nano  # Set preferred editor
[extensions]
color =        # Enable colored output
```

5. Best Practices

1. **Commit Atomic Changes**: Each commit should solve one logical task.
2. **Write Meaningful Messages**: Explain *why* (not just *what*).
3. **Pull Before Push**: Avoid merge conflicts by syncing frequently.
4. **Use Branches for Experiments**: Isolate unstable work from main.
5. **Review History Before Sharing**: hg outgoing + hg diff -r tip

Why This Matters

Mercurial's daily commands prioritize **safety** and **reproducibility**:

- No data loss: revert > undo
- Full audit trail: Annotate/log track *every* change
- Conflict resolution: Explicit workflows prevent accidental overwrites
- Distributed model: Work offline without compromising collaboration

💡 **Key Insight**: Mercurial treats all actions as *immutable history additions*. Even "undo" operations create new commits, preserving forensic integrity.

1. Core Collaboration Models

A. Repository Relationships

Type	Command	Workflow
Centralized	hg clone central-repo	Single "source of truth" repo
Peer-to-Peer	hg clone peer-repo	Direct sharing between equal repos
Hierarchical	hg pull upstream	Maintain personal clone + contribute upstream

B. Key Concepts

- **Heads**: Unmerged branch endpoints in repository history
- **Changegroups**: Bundles of changesets transferred during push/pull
- **Phases**:
  - **Public**: Changesets visible to others (immutable)
  - **Draft**: Local changes (can be modified)
  - **Secret**: Not shared (e.g., unfinished work)

2. Essential Collaboration Commands

A. Sharing Workflow

```
hg pull [source]    # Fetch changes from another repo
```

```
hg update          # Apply changes to working directory
```

```
# ... make local changes ...
```

```
hg commit
```

```
hg push [destination]  # Send changes to another repo
```

- Critical Flags:
  - **hg pull -u: Pull + immediate update**
  - **hg push --new-branch: Push unnamed branches**
  - **hg push -f: Force push (avoid unless necessary)**

B. Change Inspection

Command	Purpose
hg incoming	Preview changes before pulling
hg outgoing	Preview changes before pushing
hg heads	List all branch tips
hg heads --closed	Include closed branches

3. Branch Management in Teams

A. Named vs. Anonymous Branches

Type	Visibility	Use Case
Named	Globally visible	Long-term features/releases
Anonymous	Local until pushed	Short-lived experiments

B. Branch Operations

- `hg branch new-feature    # Create named branch`

- `hg commit -m "Start feature"`
- `hg branches          # List all branches`
- `hg update main        # Switch branches`
- `hg merge new-feature    # Merge branches`
- `hg branch --close new-feature # Close obsolete branch`

4. Advanced Collaboration Techniques

A. Handling Divergent History

- **Scenario**: Multiple developers commit to same branch
- **Solution**:
- `hg pull      # Fetch others' changes`
- `hg merge      # Merge remote changes into local`
- `# Resolve conflicts if needed`
- `hg commit -m "Merge"`
- `hg push`

B. Cherry-Picking Changes

```
hg graft -r 1234  # Copy specific changeset to current branch
```

C. Sharing Patches

1. **Export**:
   hg export -r 1234 > patch.diff
2. **Import**:
   hg import patch.diff

5. Conflict Resolution Workflow

1. **Detect Conflicts**:
2. `hg merge      # Aborts with conflict list`
3. **Inspect Conflicts**:
4. `hg resolve --list  # Show conflicted files`
5. **Resolve Manually**:
   - Edit files with conflict markers (<<<<<<<, =======, >>>>>>>)
6. **Mark Resolved**:
7. `hg resolve --mark file.txt`
8. **Complete Merge**:
9. `hg commit -m "Merge"`

6. Remote Repository Protocols

Protocol	Example URL	Authentication
SSH	ssh://user@server/repo	Key-based
HTTP/HTTPS	https://server/repo	Basic auth/cookies
Local	/path/to/repo	File permissions

Concept	Simple Explanation
.hgignore	Tells Mercurial which files to ignore
hg add	Adds files to be tracked
hg status	Shows modified/untracked files
hg commit	Saves changes to repo history
hg push/pull	Syncs your repo with another
hg tag	Marks a specific revision
branch/bookmark	Create development lines

7. Filesystem Quirks

1. **Special Characters**:
   - Escape spaces: hg add "file with spaces.txt"
   - Handle symbols: hg add 'file@#$.txt'
2. **Reserved Names** (Windows):
   Avoid CON, PRN, AUX, NUL
3. **Path Length** (Windows):
   Max 260 chars (enable long paths in registry)

8. Critical Commands Cheat Sheet

Command	Purpose
hg files "*.py"	List versioned Python files
hg add "img/*.png"	Add all PNGs in img/
hg remove -I 'patterns.txt'	Remove files matching patterns file
hg status -X '.hgignore'	Show status excluding .hgignore
hg grep -r 0:tip "TODO" --include "*.py"	Search "TODO" in Python files across history

Branch Types & Use Cases

Branch Type	Purpose	Lifespan
main (default)	Stable development	Permanent
Feature Branch	Build new features	Short-term
Release Branch	Stabilize code for release	Long-term
Hotfix Branch	Emergency production fixes	Very short (merge & close

Creating a New Release

hg update main   # Start from main

hg branch release-1.0 # Create release branch

hg commit -m "Create release branch"

hg tag v1.0.0 -m "Version 1.0.0"

hg push --branch release-1.0 --tags

Now you’ve created a tagged, stable, production-ready version

Post-Release Maintenance

- Fix bugs in the release branch:

hg update release-1.0

fix code

hg commit -m "Fix crash in auth module"

hg tag v1.0.1 -m "Patch release"

- Merge fixes into main:

hg update main

hg merge release-1.0

hg commit -m "Merge release fixes into main"

🌱 Feature Development Workflow

A. Create and Use Feature Branch

hg update main

hg branch feature-payments

develop code

hg commit -m "Add payment gateway support"

Merge and Close

```
hg update main
hg merge feature-payments
hg commit -m "Merge payment feature"
hg update feature-payments
hg commit --close-branch -m "Feature done"
```

Hotfix Workflow (Emergency Fix)

```
hg update v1.0.0
hg branch hotfix-login
# apply fix
hg commit -m "Fix login bug"
```

Merge to release and main:

```
hg update release-1.0
hg merge hotfix-login
hg commit -m "Merge hotfix to release"
hg tag v1.0.2 -m "Hotfix release"

hg update main
hg merge hotfix-login
hg commit -m "Merge hotfix to main"
hg commit --close-branch -C hotfix-login
```

Key Commands

Command	Purpose
hg branches	Show active branches
hg branches --closed	Include closed ones
hg heads	See tips of branches
hg update -c <branch>	Discard local changes on switch

Critical Commands Cheat Sheet

Task	Command Example
Create branch	hg branch feature-xyz
Switch branch	hg update release-1.0
Close branch	hg commit --close-branch -m "Done"
Create tag	hg tag -m "Stable release" v2.1.0
Merge	hg merge feature-xyz
Push specific	hg push --branch release-1.0

Key Insight

Mercurial **treats branches as permanent history**, not just pointers.
It's like a historical timeline with **labels and context**.
Every action is **reversible, trackable, and safe**.

2. Working Directory Mistakes

A. Discard Uncommitted Changes

- `hg revert <file>        # Revert single file`
- `hg revert -a            # Revert all changes`
- `hg revert -d            # Revert to last commit (discard new files)`
- `hg clean                # Reset entire working directory (requires purge extension)`

💡 **Key Insight**: revert only affects working directory – committed changes remain safe

B. Undo Adds/Removes

- `hg forget <file>        # Unstage added file (keeps working copy)`
- `hg addremove            # Smart re-add after mistaken removal`

3. Fixing Committed Mistakes

A. Amend Last Commit

```
# Change files/staging
```

```
hg commit --amend       # Replace last commit
```

- **Changes**:
    - Modifies commit message/content
    - Original commit becomes hidden (not destroyed)

B. Reverse Specific Commits

```
hg backout -r <REV>     # Create inverse commit to undo REV
```

Example:

```
hg backout -r 1234 -m "Undo feature X"
```

C. Historical Correction

```
hg graft -r <REV>       # Re-apply good commit
```

```
hg strip -r <REV>       # Remove bad commit (requires mq extension)
```

4. Advanced Correction Tools

A. MQ Extension (Mercurial Queues)

```
hg qimport -r <REV>     # Convert commit to patch
```

```
hg qpop                 # Unapply patch
```

```
# Edit files
```

```
hg qrefresh             # Update patch
```

```
hg qpush            # Reapply patch
```

```
hg qfinish          # Convert patch to permanent commit
```

B. Histedit Extension

```
hg histedit -r <REV>    # Interactive rebase
```

Operations:

- pick (keep commit)
- edit (pause for changes)
- drop (remove commit)
- fold (combine commits)

5. Recovery Techniques

A. Find Lost Commits

bash

```bash
hg log -r "extinct()"    # Show hidden commits
```

```bash
hg log -r "secret()"    # Show unpublished work
```

B. Restore Stripped Commits

```
hg recover          # Restore from transaction journal
```

Recovery sources:

1. .hg/strip-backup/
2. .hg/journal/

6. Critical Scenarios & Solutions

Mistake	Solution	Risk Level
Committed wrong files	hg commit --amend	Low
Pushed bad commit	hg backout + new commit	Medium
Accidentally stripped commit	hg unbundle .hg/strip-backup/<bundle>	High
Wrong branch commit	hg graft -r <REV> -b correct-branch	Medium
Sensitive data in history	hg convert (filter repo)	Very High

7. Safety Mechanisms

1. **Phases System:**
2.
3. hg phase -r <REV> -p   # Mark as public (immutable)
4. hg phase -r <REV> -d   # Mark as draft (modifiable)
5. **Undo History:**
6.
7. hg rollback   # Undo last *transaction* (deprecated)
8. **Backups:**
9.
10. hg clone -r <REV> . ../backup # Create point-in-time clone

8. Best Practices

1. **Verify Before Sharing:**
2. bash
3. hg outgoing   # Preview before push
4. **Use Draft Phase:**
5. ini
6. [phases]
7. publish = false   # Default to draft phase
8. **Atomic Corrections:**
   - One fix per backout/amend
9. **Document Changes:**
10. Download
11. hg commit -m "FIX: Revert bad data import (backout:1234)"

Why Mercurial's Approach Wins

1. **Non-Destructive Workflow:**
   - Original commits preserved for audit trails

2. **Explicit Operations**:
   - No automatic history rewriting
3. **Recovery Fallbacks**:
   - Built-in backups for critical operations
4. **Enterprise Safety**:
   - Compliance with change management policies

💡 **Key Insight**: Mercurial treats "mistakes" as *new version states* rather than history deletion. This ensures traceability while allowing correction.

1. Core Templating System

A. Template Fundamentals

- **Syntax**: {keyword} or {expression|filter}
- **Execution**:
-
- `hg log --template "{date|isodate}: {author}\n"`

B. Key Components

Element	Example	Output	
Keywords	{node}	b4d73b75e0a3 (full hash)	
Filters	`{node	short}`	b4d73b75 (abbreviated)
Functions	`{date	isodate}`	2023-04-15 12:30:00 +0200
Conditionals	{if(branch, 'Branch: ')}	Only shows if branch exists	

2. Essential Keywords

Keyword	Description	Command Context
{rev}	Revision number	log, status, heads
{node}	Full changeset hash	All history commands
{author}	Committer name + email	log, annotate
{desc}	Full commit message	log
{files}	List of changed files	log, status
{branches}	Branch names	log
{bookmarks}	Bookmark names	log
{tags}	Tags associated with changeset	log

3. Powerful Filters

Filter	Function	Example	
short	Abbreviate hash to 12 chars	`{node	short} →b4d73b75e0a3`
firstline	Extract first line of description	`{desc	firstline}`
person	Extract username from author	`{author	person}`
email	Extract email from author	`{author	email}`
date	Format date (with format string)	`{date	date}'%Y-%m-%d'`
fill	Word-wrap text to specified width	`{desc	fill68}`
strip	Remove trailing newlines	`{desc	strip}`
urlescape	URL-encode strings	`{author	urlescape}`

4. Template Functions

A. String Manipulation

`{rev}: {desc|firstline|lower}`

→ 42: fix login bug

B. Conditional Logic

template

Copy

Download

```
{if(branch, 'Branch: {branch}')}
```

→ Only shows branch if exists

C. Join Operations

template

```
{files % '- {file}\n'}
```

→ Lists files with bullets

5. Predefined Styles

Style	Command	Use Case
default	hg log	Standard detailed output
compact	hg log --style compact	Short revision + message
changelog	hg log --style changelog	Emulates traditional changelog
xml	hg log --style xml	Machine-readable output
json	hg log --style json	JSON formatted output

6. Custom Style Configuration

A. Global Styles (~/.hgrc)

ini

```ini
[templates]
my_style = "{rev}:{node|short} {author|person}\n  {desc|firstline}\n\n"
```

Usage: hg log --style my_style

B. Command-Specific Overrides

ini

Copy

Download

```ini
[alias]
```

```
ls = log --style compact -l 10
```

7. Advanced Formatting

A. Multi-Line Templates

ini

```ini
[templates]

release = "Version {latesttag}:\n{desc|fill76}\nChanges:\n{files % ' - {file}\n'}"
```

B. Color Output

ini

```ini
[extensions]

color =


[color]

status.modified = green bold

log.changeset = yellow
```

Usage: hg log --color always

8. Practical Examples

A. Release Notes Generator

bash

Copy

Download

```bash
hg log -r v1.0:v2.0 --template \

"{rev}: {author|person}\n  {desc|fill76|strip}\n  Files: {files % '{file}, '}\n\n"
```

B. Machine-Readable Output

bash

```bash
hg log -l1 --template '{"rev": {rev}, "node": "{node}", "author": "{author|json}"}\n'
```

→ {"rev": 42, "node": "b4d73b75e0a3", "author": "John Doe <john@example.com>"}

C. Branch Visualization

Download

```
hg log -G --template "{label('changeset.{phase}', rev)} {branch} {desc|firstline}\n"
```

9. Best Practices

1. **Reusability**: Define styles in ~/.hgrc instead of command-line templates
2. **Readability**:
   - Wrap long descriptions with fill
   - Use colors sparingly
3. **Portability**:
   - Avoid platform-specific characters
   - Use \n for newlines
4. **Escape Sequences**:
   - \n → Newline
   - \t → Tab
   - \\ → Backslash
   - \" → Double quote

Why Templating Matters

1. **Automation**: Generate release notes/CHANGELOGs directly from history
2. **Integration**: Create machine-readable output for CI/CD pipelines
3. **Clarity**: Highlight critical information (branches, tags, authors)
4. **Consistency**: Enforce team-wide formatting standards
5. **Efficiency**: Replace custom scripts with built-in templating

💡 **Pro Tip**: Combine with hg -v for template debugging (shows available keywords)

ini

```ini
# Ultimate Productivity Template

[alias]

mylog = log --template "{label('yellow',rev)} {label('green', author|person)}: {desc|firstline}\n  {label('cyan', files % '{file} ')}\n" --color always
```

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.
4. [hooks]
5. precommit = ./check_policy.py # Custom compliance checks
6. **LDAP Authentication**:
7. ini
8. [auth]
9. company.prefix = https://hg.company.com
10. company.username = DOMAIN\user
11. **Audit Logging**:
12. bash
13. 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

EdenSCM is a **cross-platform**, highly **scalable source control system** designed for **very large monorepos** containing millions of files and commits. It was developed at Facebook (now Meta) and also known as **Sapling.**

- It maintains a **Mercurial-like user experience** via the eden CLI
- It is **not purely distributed**—it uses **server-client coordination** for efficiency in large codebases

EdenSCM treats your repo as a "virtual filesystem" with smart syncing — not as a static local copy. It's a paradigm shift that prioritizes **speed**, **scale**, and **efficiency**, perfect for **internal mega-codebases**, but not meant for general open-source use.

Architecture & Key Components

EdenSCM comprises three main components

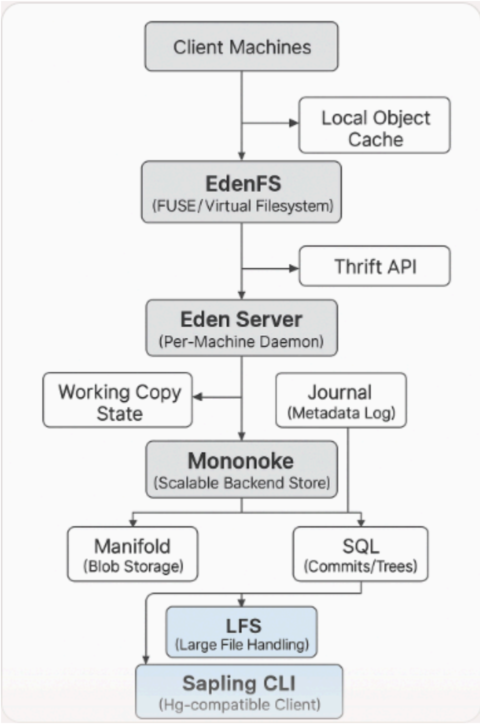
- **eden CLI**:
  - User-facing command tool, Mercurial-inspired (e.g., eden commit, eden update).
- **Mononoke** (server-side):
  - Rust-based system managing repository data.
  - Enables **on-demand fetch**—clients download only what is needed, not the entire history .
- **EdenFS** (virtual filesystem):
  - A FUSE-like layer that populates files only when accessed.
  - Provides sparse-checkout behavior transparently, with fast status queries and Watchman integration.

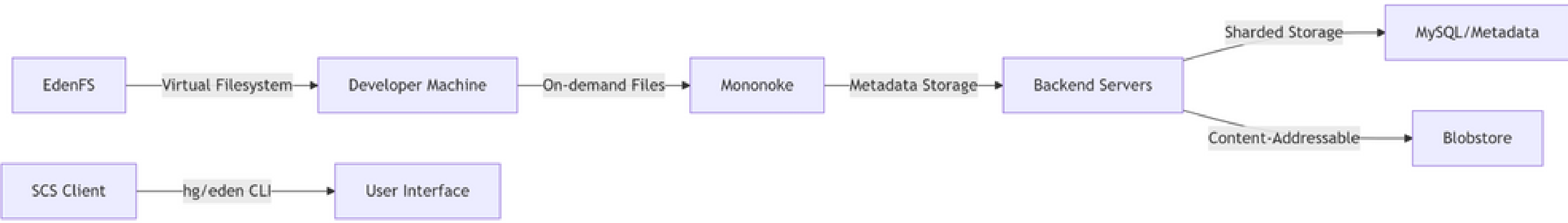
(Formerly Facebook's Source Control System)

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





How does EdenSCM achieve O(1) checkouts?

"Via FUSE-based virtual filesystem. Only fetches file contents on first access, while directory structure is materialized instantly using pre-fetched metadata."

Handling large binaries?

"Integrated LFS with asynchronous upload. Client commits reference LFS pointers, while binaries replicate globally via Manifold within 60s.

Disaster recovery?

"Multi-region Mononoke clusters with CRDT-based eventual consistency. Journal replays reconstruct state within 5 mins of outage.

Compare to Git-LFS/Git-VFS?

"EdenSCM natively handles scale without per-repo configs. Git-LFS requires manual setup; Git-VFS lacks Watchman integration for instant status."

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 Differences (Simplified)

Feature	**EdenSCM**	**Git**	**Mercurial**
Checkout Speed	⚡ Instant (loads only used files)	🐢 Slower (downloads all files)	🐢 Slower (downloads all files)
Disk Usage	🧠 Smart — only stores needed files	🚫 High — stores everything	🚫 High — stores everything
Clone Time	⏱️ Fast (just metadata)	🕐 Slower (clones entire repo)	🕐 Slower
File Access	🔄 On demand (lazy loading)	✅ Immediate (everything is present)	✅ Same
Merge Handling	👍 Good for known branches	💪 Powerful, but messy history possible	✅ Clean & simple merges
Scaling to Large Repos	✅ Built for scale	❌ Needs tooling help	⚠️ Better, but not optimized
Storage System	🛠️ Mononoke backend	Local .git/ folder	Local .hg/ folder

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

 # 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

Checkout branch (instant)

eden checkout feature/new-design

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

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

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.

Buck2

Buck2 is a **build system** — a tool used to **compile code**, **build binaries**, and **run tests** efficiently, especially in **large and complex software projects**. It was developed by **Meta (Facebook)** to scale and speed up builds across their massive codebase.

Buck2: Core Concepts for Interview Preparation

Buck2 is Meta's next-generation, open-source build system designed for **extreme speed, scalability, and correctness**. It succeeds Buck1, with a complete rewrite in Rust and enhanced Starlark integration. Below is a structured breakdown:

Definition

Buck2 is a **monorepo-optimized build tool** that automates compiling, testing, and packaging code. It emphasizes:

- **Hermeticity:** Reproducible builds (same inputs → same outputs).
- **Incrementality:** Minimal rebuilds via precise dependency tracking.
- **Parallelization & Caching:** Maximizes speed through concurrency and local/remote caching.

What Buck2 Does

Task	Example
Compiling source code	Compiling C++, Rust, Java, Python, etc. into binaries or libraries
Managing dependencies	Only rebuilding what changed instead of everything
Testing automation	Running unit/integration tests as part of the build
Remote execution	Sending build tasks to fast remote servers (for speed)
Caching	Reusing previous build outputs to avoid redundant work
Hermetic builds	Ensures builds are reproducible and consistent on all machines

Why Use Buck2?

✅ When you have:

- A **large monorepo** (millions of files)
- **Multiple languages** (e.g. Python, Java, Rust, C++)
- Need **fast incremental builds**
- Require **reliable, reproducible builds**
- Want to **run builds in the cloud** or **parallelized environments**

At Meta (Facebook):

- Buck2 handles **millions of builds per day**
- Used for **Facebook, Instagram, WhatsApp** backend/frontend code
- Speeds up build times by **2–10×** compared to older systems

🧵 In Simple Words:

Buck2 is like a **super-smart assistant** that watches your code and only **rebuilds what's needed**, **runs tests**, and **ships code faster**, all while working with **any programming language** in your repo.

How Buck2 Works – Workflow

1. **Write BUCK files** (in Starlark) defining build targets.

2. **Run**: buck2 build //:app.
3. **Parsing & Configuring**: Core parses BUCK; builds **configured target graph**.
4. **Analysis**: Executes target rules, producing **actions** and **providers** (output declarations).
5. **Execution Graph**: Dependencies translated into build DAG.
6. **Execution Phase**:
   - Compute **content hashes** for each action.
   - Check remote cache → reuse outputs if available.
   - Otherwise, execute locally/remotely (or race both), then cache results.
7. **Materialization**: Lazy download of artifacts to buck-out as needed.
8. **Testing**: If buck2 test, runs test actions via TPX

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
```

Real-World Example Workflow

1. Clone the monorepo
2. Create BUCK file with java_binary()
3. Build with buck2 build //app:my_app
4. Run with buck2 run //app:my_app
5. Add tests → buck2 test //app:test_suite
6. Push → CI runs buck2 test and buck2 build

ck2 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 **build system generator**. It doesn't build software itself, but generates native build files (like Makefiles, Ninja, Visual Studio projects) for various platforms and compilers from a single configuration file (CMakeLists.txt).

- Created by **Kitware**
- Cross-platform: Supports Linux, Windows, macOS, Android, iOS
- Language-independent: Works with C, C++, Fortran, CUDA, etc.

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**

Key Components

Component	**Description**
CMakeLists.txt	**Primary configuration file describing the build process**
Generator	**Converts CMake config into native build files**
Cache	**Stores build settings (e.g., paths, flags) in CMakeCache.txt**
Targets	**Logical units like libraries or executables**
Modules	**Scripts that provide functionality (e.g., FindBoost.cmake)**
Toolchains	**Define the compiler and platform (e.g., for cross-compiling)**

Why CMake?
Developers write CMakeLists.txt once, and CMake generates platform-specific build files.

CMake is a *meta-build* tool: It creates build scripts for tools like Make/Ninja. Make/Ninja are *build executors* that run those scripts."

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

Key Components

Component	Description
CMakeLists.txt	Primary configuration file describing the build process
Generator	Converts CMake config into native build files
Cache	Stores build settings (e.g., paths, flags) in CMakeCache.txt
Targets	Logical units like libraries or executables
Modules	Scripts that provide functionality (e.g., FindBoost.cmake)
Toolchains	Define the compiler and platform (e.g., for cross-compiling)

🎯 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

What is the purpose of CMakeLists.txt?

Definition:
The CMakeLists.txt file is CMake's **project configuration manifest**. It declares:

- Build targets (executables, libraries).
- Dependencies (internal/external).

- Compiler flags, include paths, and installation rules.
- Tests, packaging, and cross-platform conditions.

The CMakeLists.txt file is CMake's project manifest that contains directives and instructions CMake uses to generate native build tools like Makefiles, Ninja, or Visual Studio project files.

Line	**Purpose / Explanation**
cmake_minimum_required(VERSION 3.15)	Ensures CMake version is at least 3.15. Prevents compatibility issues.
project(MyExecutableApp VERSION 1.0 LANGUAGES CXX)	Defines the project name (MyExecutableApp), version (1.0), and programming language (C++).
set(CMAKE_CXX_STANDARD 17)	Sets the C++ standard to C++17.
set(CMAKE_CXX_STANDARD_REQUIRED ON)	Enforces that C++17 must be used (no fallback).
add_executable(my_app src/main.cpp)	Adds a build target named my_app using the source file src/main.cpp.
target_include_directories(my_app PRIVATE ${PROJECT_SOURCE_DIR}/include)	Adds the include/ directory (if it exists) to the compiler's header search path (for this target only).
# target_link_libraries(my_app PRIVATE some_library)	*(Optional)*: Allows linking external libraries (e.g., pthread, Boost, etc.).
install(TARGETS my_app DESTINATION bin)	Defines install rules: puts the built my_app binary into a bin/ folder on make install.

🛠️ 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

---

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
#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

```cpp
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

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

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

🧼 Output in XML (CI/CD friendly)

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

📦 Integrating with CMake

CMakeLists.txt

cmake

```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  |
+--------------------------+
```

LLVM?

✅ Definition

LLVM (Low-Level Virtual Machine) is a **modular, reusable compiler infrastructure** designed for **compile-time, link-time, run-time, and idle-time optimization** of programs.

- It is not just a virtual machine—it's a **toolchain and backend framework** for building compilers, code analyzers, debuggers, and JIT engines.

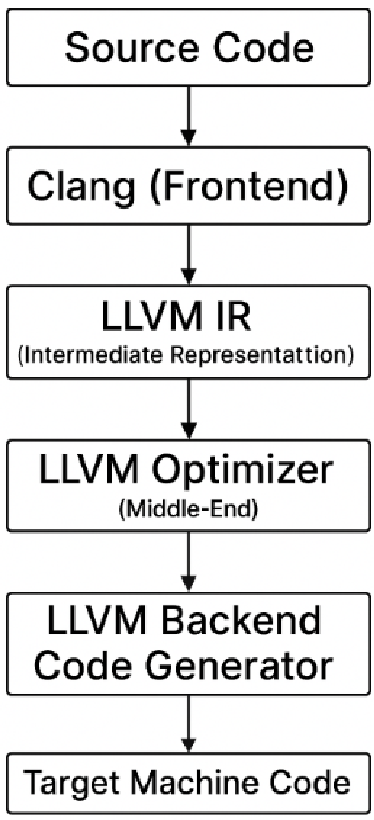
Clang is a **C, C++, and Objective-C front end** for LLVM. It parses source code and generates LLVM Intermediate Representation (IR).

- Clang is designed to be:
  - Fast
  - Modular
  - User-friendly with GCC-compatible flags
  - Better diagnostics and tooling support

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).



Stage	Role
Clang	Frontend: Parses code and produces LLVM IR
LLVM IR	Typed, platform-independent low-level code
Optimizer	Applies code optimizations (dead code elimination, inlining, etc.)
Backend	Converts LLVM IR to native assembly for various targets

LLVM Key Components

Component	Description
Clang	C/C++/Obj-C frontend → LLVM IR
LLVM IR	Language-independent intermediate code
opt	LLVM Optimizer: applies IR-level optimizations
llc	Converts LLVM IR to machine code
clang++	Compiles C++ using Clang frontend and LLVM backend
llvm-as/llc	Assembler/Disassembler for LLVM IR
libclang	C API for tooling
clangd	Language server for IDE tooling
LLVM Passes	Optimization passes (inline, loop unroll, etc.)

"LLVM uses a modular, library-based design with a unified IR, while GCC is a monolithic compiler. Clang (LLVM) offers faster compilation, better diagnostics, and permissive licensing."

LLVM IR (Intermediate Representation) is a **platform-independent, low-level, RISC-like instruction set** that acts as the universal code representation between frontends (e.g., Clang) and backends (e.g., x86, ARM).

Compare Clang with GCC

Aspect	Clang (LLVM)	GCC
Architecture	Modular (frontend + optimizer + backend)	Monolithic
Compile Speed	1.5–2× faster (avg.)	Slower
Memory Usage	Lower footprint	Higher (especially for templates)
Diagnostics	Clear, colorized, actionable errors	Less user-friendly
Sanitizers	ASan, UBSan, TSan, MSan	Limited support
License	Apache 2.0 (business-friendly)	GPL (copyleft)
C++ Standards	Faster adoption of C++20/23	Slower to implement new features
Debug Info	Better DWARF5 support (GDB, LLDB)	Mature but less optimized

🧩 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)

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

clang-tidy memory_utils.c -- -Iinclude/

3. Format All Source Code in Directory

```
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 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**

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

Key Components Explained:

1. **Dev Workflow** (Blue boxes):
   - Dev writes code: Initial coding phase
   - arc diff: Creates reviewable diff
   - arc land: Commits approved code
2. **Review Process**:
   - Phabricator Differential: Web-based code review tool
   - Approval loop: Iterative review until approval
3. **CI/CD Pipeline**:
   - Harbormaster: Phabricator's CI system
   - Automated build/test execution
   - Pass/Fail decision gate
4. **Outcome Paths**:
   - ✅ **Success**: Code deployed after passing CI
   - ❌ **Failure**: Alert developer with auto-revert

Real-World Example Flow:

1. Developer at Meta works on Facebook News Feed feature
2. Creates diff: arc diff --create
3. Team reviews on Phabricator (comments, requests changes)
4. After approval: arc land
5. Harbormaster triggers:
   - Build compilation
   - Unit/integration tests
   - iOS/Android compatibility checks
6. On success: Automatically deployed to canary servers
7. On failure: Immediate Slack alert to developer

Advantages of This Workflow:

1. **Pre-commit Reviews**: Ensures quality before merging

2. **Automated Verification**: Catches regressions pre-deployment
3. **Traceability**: Full audit trail from code to deployment
4. **Developer Velocity**: Parallel review/CI processes
