build-systems

- Requirements
  - Platform support
  - Auto-detection of supported features
  - Ease of use
  - IDE support
  - Out-of-tree builds
  - Cross-platform builds
  - Language support
  - Test integration
- Comparison
  - GNU Make
  - autoconf
  - CMake
  - Meson

The build system is the primary way for both developers and system integrators to interact with the Git project. As such, being easy to use and extend for those who are not directly developing Git itself is just as important as other requirements we have on any potential build system.

This document outlines the different requirements that we have for the build system and then compares available build systems using these criteria.

Requirements

The following subsections present a list of requirements that we have for any potential build system. Sections are sorted by decreasing priority.

Platform support

The build system must have support for all of our platforms that we continually test against as outlined by our platform support policy. These platforms are:

- Linux
- Windows
- macOS

Furthermore, the build system should have support for the following platforms that generally have somebody running test pipelines against regularly:

- AIX
- FreeBSD
- NetBSD
- NonStop
- OpenBSD

The platforms which must be supported by the tool should be aligned with our platform support policy (see platform-support.adoc).

Auto-detection of supported features

The build system must support auto-detection of features which are or aren't available on the current platform. Platform maintainers should not be required to manually configure the complete build.

Auto-detection of the following items is considered to be important:

- Check for the existence of headers.
- Check for the existence of libraries.
- Check for the existence of executables.
- Check for the runtime behavior of specific functions.
- Check for specific link order requirements when multiple libraries are involved.

Ease of use

The build system should be both easy to use and easy to extend. While this is naturally a subjective metric it is likely not controversial to say that some build systems are considerably harder to use than others.

IDE support

The build system should integrate with well-known IDEs. Well-known IDEs include:

- Microsoft Visual Studio
- Visual Studio Code
- Xcode

There are four levels of support:

- Native integration into the IDE.
- Integration into the IDE via a plugin.
- Integration into the IDE via generating a project description with the build system.
- No integration.

Native integration is preferable, but integration via either a plugin or by generating a project description via the build system are considered feasible alternatives.

Another important distinction is the level of integration. There are two features that one generally wants to have:

- Integration of build targets.
- Automatic setup of features like code completion with detected build dependencies.

The first bullet point is the bare minimum, but is not sufficient to be considered proper integration.

Out-of-tree builds

The build system should support out-of-tree builds. Out-of-tree builds allow a developer to configure multiple different build directories with different configuration, e.g. one "debug" build and one "release" build.

Cross-platform builds

The build system should support cross-platform builds, e.g. building for arm on an x86-64 host.

Language support

The following languages and toolchains are of relevance and should be supported by the build system:

- C: the primary compiled language used by Git, must be supported. Relevant toolchains are GCC, Clang and MSVC.
- Rust: candidate as a second compiled language, should be supported. Relevant toolchains is the LLVM-based rustc.

Built-in support for the respective languages is preferred over support that needs to be wired up manually to avoid unnecessary complexity. Native support includes the following features:

- Compiling objects.
- Dependency tracking.
- Detection of available features.
- Discovery of relevant toolchains.
- Linking libraries and executables.
- Templating placeholders in scripts.

Test integration

It should be possible to integrate tests into the build system such that it is possible to build and test Git within the build system. Features which are nice to have:

- Track build-time dependencies for respective tests. Unit tests have different requirements than integration tests.
- Allow filtering of which tests to run.
- Allow running tests such that utilities like `test_pause` or `debug` work.

Comparison

The following list of build systems are considered:

- GNU Make
- autoconf
- CMake
- Meson

GNU Make

- Platform support: ubiquitous on all platforms, but not well-integrated into Windows.
- Auto-detection: no built-in support for auto-detection of features.
- Ease of use: easy to use, but discovering available options is hard. Makefile rules can quickly get out of hand once reaching a certain scope.
- IDE support: execution of Makefile targets is supported by many IDEs
- Out-of-tree builds: supported in theory, not wired up in practice.
- Cross-platform builds: supported in theory, not wired up in practice.
- Language support:
- C: Limited built-in support, many parts need to be wired up manually.
- Rust: No built-in support, needs to be wired up manually.
- Test integration: partially supported, many parts need to be wired up manually.

autoconf

- Platform support: ubiquitous on all platforms, but not well-integrated into Windows.
- Auto-detection: supported.
- Ease of use: easy to use, discovering available options is comparatively easy. The autoconf syntax is prohibitively hard to extend though due to its complex set of interacting files and the hard-to-understand M4 language.
- IDE support: no integration into IDEs at generation time. The generated Makefiles have the same level of support as GNU Make.
- Out-of-tree builds: supported in theory, not wired up in practice.
- Cross-platform builds: supported.
- Language support:
- C: Limited built-in support, many parts need to be wired up manually.
- Rust: No built-in support, needs to be wired up manually.
- Test integration: partially supported, many parts need to be wired up manually.

CMake

- Platform support: not as extensive as GNU Make or autoconf, but all major platforms are supported.
- AIX
- Cygwin
- FreeBSD
- Linux
- OpenBSD
- Solaris
- Windows
- macOS

- Ease of use: easy to use, discovering available options is not always trivial. The scripting language used by CMake is somewhat cumbersome to use, but extending CMake build instructions is doable.
- IDE support: natively integrated into Microsoft Visual Studio. Can generate project descriptions for Xcode. An extension is available for Visual Studio Code. Many other IDEs have plugins for CMake.
- Out-of-tree builds: supported.
- Cross-platform builds: supported.
- Language support:
- C: Supported for GCC, Clang, MSVC and other toolchains.
- Rust: No built-in support, needs to be wired up manually.
- Test integration: supported, even though test dependencies are a bit cumbersome to use via "test fixtures". Interactive test runs are not supported.

Meson

- Platform: not as extensive as GNU Make or autoconf, but all major platforms and some smaller ones are supported.
- AIX
- Cygwin
- DragonflyBSD
- FreeBSD
- Haiku
- Linux
- NetBSD
- OpenBSD
- Solaris
- Windows
- macOS
- Ease of use: easy to use, discovering available options is easy. The scripting language is straight-forward to use.
- IDE support: Supports generating build instructions for Xcode and Microsoft Visual Studio, a plugin exists for Visual Studio Code.
- Out-of-tree builds: supported.
- Cross-platform builds: supported.
- Language support:
- C: Supported for GCC, Clang, MSVC and other toolchains.
- Rust: Supported for rustc.
- Test integration: supported. Interactive tests are supported starting with Meson 1.5.0 via the `--interactive` flag.

□□□ #2
□ □□□ □□□ 2026-10-31 20:16:19 UTC
□ □□□ □□□ 2026-10-31 20:16:20 UTC