Setting Up Conan and CMake for PhysKit (Windows & macOS)

In this guide, we will walk through installing the Conan package manager, configuring a C++26-compliant Conan profile, installing dependencies via Conan, setting up CMake presets for the PhysKit project, and integrating everything with VSCode (including compile commands for clangd). Both Windows and macOS development environments are covered. Short, step-by-step instructions and troubleshooting tips are provided to ensure a smooth setup.

Installing the Conan Package Manager

Conan is a C++ package manager that we will use to manage PhysKit's dependencies. The recommended way to install Conan is via Python's pip, or via Homebrew on macOS:

- **Windows:** Ensure you have Python 3 installed, then install Conan using pip. In a Command Prompt or PowerShell:

  ```
  pip install conan
  ```

  (If you get a permission error, add `--user` or use an elevated prompt.) This will install the `conan` command globally[1][2]. After installation, verify by running `conan --version` in a new terminal.

- **macOS:** You can use Homebrew to install Conan directly:

  ```
  brew update && brew install conan
  ```

  This will install Conan along with its Python dependency[3]. Alternatively, pip works on macOS as well (`pip install conan`). After installation, you might need to run `source ~/.profile` or restart your terminal so that `conan` is on your PATH[4][5]. Verify by running `conan --version`.

Tip: Conan updates frequently. You can upgrade an existing installation with `pip install --upgrade conan` if needed[6].

Creating a C++26-Compatible Conan Profile

Conan uses *profiles* to define your build settings (OS, compiler, architecture, etc.). We need a profile that specifies a compiler capable of C++26 features. Apple's default Clang (the one included with Xcode) lags behind upstream Clang and may not fully support C++26[7], so on macOS we'll prefer the latest LLVM Clang. On Windows, we'll use Visual Studio 2022 (MSVC) in C++latest mode or an alternative modern compiler.

Steps to create/configure a profile:

1. **Auto-detect a baseline profile:** Run `conan profile detect --force` to let Conan generate a profile with your current environment's defaults. This will create (or overwrite) a default profile file (often in `~/.conan2/profiles/default` for Conan 2.x) with detected settings. For example, on a Mac it might detect Apple-Clang 14 and set `compiler.cppstd=gnu17` by default[8]. (The auto-detection tends to default to C++17 if not specified, so we'll adjust it next.)

2. **Locate and edit the profile:** Open the generated profile (e.g. `default` profile). Profiles are simple text files with sections like `[settings]`, `[env]`, etc. We need to ensure the settings reflect a C++26-capable compiler and standard:

3. **macOS (LLVM Clang):** Under `[settings]`, set `compiler` to **clang** (not apple-clang) to use the LLVM toolchain from Homebrew, and set a recent version number (e.g., `compiler.version=21` if you installed LLVM 21, or higher if available). Also set `compiler.libcxx=libc++` (for Clang on macOS) and `compiler.cppstd=26`. For GNU extensions, you can use `gnu23`/`gnu26` (e.g., `compiler.cppstd=gnu23`)[9]. In summary:

   ```
   [settings]
   os=Macos
   arch=x86_64  # or armv8 for Apple Silicon
   compiler=clang
   compiler.version=21    # use the actual llvm version installed
   compiler.libcxx=libc++
   compiler.cppstd=26     # target C++26
   build_type=Release     # default build type (can be overridden)
   ```

   Ensure you have installed the corresponding LLVM toolchain (e.g. via Homebrew) and that the `clang++` for that version is accessible. Apple Clang (Xcode) does **not** support `-std=c++26` yet, whereas Homebrew's LLVM Clang does support the latest standards. [10][7].

4. **Windows (Visual Studio/MSVC or Clang):** If using MSVC, make sure you're using a recent version that supports C++26. Set `compiler.runtime=dynamic` or `static` as needed (dynamic runtime MD is typical for applications). For the C++ standard, set `compiler.cppstd=26` [11]. Example for MSVC:

   ```
   [settings]
   os=Windows
   arch=x86_64
   compiler=msvc
   compiler.version=… # your version here
   compiler.runtime=dynamic
   compiler.cppstd=26
   build_type=Release
   ```

This tells Conan that you'll be using Visual C++ 2022 with C++26. *If using an alternative compiler on Windows* (like MinGW GCC or LLVM clang-cl), adjust `compiler` and related fields accordingly (e.g., `compiler=gcc`, `compiler.version=15`, `compiler.libcxx=libstdc++`, etc., and ensure that compiler is installed and in PATH).

5. **Ensure the compiler is actually used: Important:** Setting the profile does **not** automatically switch your system compiler. Conan profiles inform Conan which compiler & standard to expect for dependency binaries, but **you must also configure CMake to use that compiler**[12][13]. This means:

6. On macOS, if you set `compiler=clang` (Homebrew LLVM), ensure that when building, you invoke that compiler. E.g., set environment variables `CC` and `CXX` to the Homebrew clang paths (e.g. `/opt/homebrew/opt/llvm/bin/clang` and `clang++`) before running CMake, or specify `-DCMAKE_C_COMPILER/-DCMAKE_CXX_COMPILER` in CMake presets. Otherwise, CMake might default to Apple clang (/usr/bin/clang). Conan won't control which compiler CMake picks – you have to do that (either via env or in the CMake preset)[12][13].
7. On Windows with MSVC, opening a Developer Command Prompt or using the VS Developer PowerShell before running Conan/CMake will ensure the MSVC compiler is available in PATH. If using VSCode, the CMake Tools extension can initialize the environment for MSVC if you select a Visual Studio kit or preset. If using a different compiler, ensure it's on PATH or provide the path in the preset.

8. When building dependencies, conan may default to use a compiler version other than the version you specified in your profile (e.., apple clang instead of homebrew clang). If this happens, add something like this:

```
[conf]
tools.build:compiler_executables={"c":"/opt/homebrew/opt/llvm/bin/clang","cpp":"/opt/homebrew/opt/llvm/bin/clang++"} # change to your compiler version paths

tools.cmake.cmaketoolchain:generator=Ninja # might not be neccessary
```

9. **(Optional) Save as a named profile:** Instead of modifying the default profile, you can create a separate profile (e.g., named `cxx26`) and use that. For example:

```
conan profile new cxx26 --detect
conan profile update settings.compiler.cppstd=26 cxx26
```

and similarly update other settings as above. Then you can use `--profile=cxx26` when invoking Conan. Storing a profile file in your repository (and using `conan config install` to distribute it) is a good practice for team consistency[14], but for now you can configure it locally.

Summary: At this point, we have Conan installed and a profile that specifies a modern compiler with C++26 support. We're now ready to install project dependencies using Conan.

Installing Dependencies via Conan

PhysKit's dependencies are defined in the `conanfile.py` at the root of the repository. The Conan recipe already includes **mp-units 2.4.0** as a requirement with certain options (notably enabling C++20 modules)[15]. We will use Conan to download/install these dependencies and generate the files needed for CMake integration.

Before proceeding, ensure you're in the project's root directory (where `conanfile.py` lives). Then follow these steps:

1. **Choose build configurations:** We want to install dependencies for both Debug and Release builds (since PhysKit will be built in those configurations). Conan can install for a given configuration (`build_type`). If you are using a single-configuration generator like Ninja or Makefiles, you'll do separate installs for each config. If using a multi-config generator (Visual Studio), one install can generate info for all configs. For clarity, we'll do two installs:

2. **Run Conan install for Release:**

   ```
   conan install . --profile=default -s build_type=Release --output-folder=build/Release --build=missing
   ```

   Replace `--profile=default` with your profile name if you created a custom one. This command will:

3. Resolve and download the required packages (e.g., mp-units) for Release.
4. Generate configuration files (due to `generators = "CMakeDeps", "CMakeToolchain"` in our conanfile[16]) under the specified `output-folder` (in this case, `build/Release/`). It will produce a **Conan toolchain file** (`conan_toolchain.cmake`) and dependency files (like `mp-units-config.cmake` via CMakeDeps).
5. By using `--build=missing`, if precompiled binaries for your exact profile aren't available, Conan will build them from source. The first run may take some time if mp-units (and its dependencies, if any) need to be built.
6. Conan 2.x will also generate CMake preset files for integration (discussed below).

If no errors occur, you should see Conan output indicating it installed or built the packages and created generator files. For example, Conan will note that it generated a toolchain file in the **generators folder**[17].

1. **Run Conan install for Debug:**

```
conan install . --profile=default -s build_type=Debug --output-
folder=build/Debug --build=missing
```

This will do the same for a Debug configuration, placing files in `build/Debug/`. Having separate output folders for each configuration keeps the generated files isolated per configuration (which is aligned with CMake's defaults under `cmake_layout`).

Note: The `cmake_layout(self)` in our conanfile instructs Conan to organize build artifacts in a `build/<config>` structure automatically[17][18]. By specifying `--output-folder=build/ConfigName`, we align with that layout. Conan will also generate **CMake Presets** as part of this process (specifically, a `CMakePresets.json` in each generator folder, and a top-level `CMakeUserPresets.json` in the project directory) to simplify using CMake with these dependencies[19].

1. **Verify Conan outputs:** After running these, check that you have:
2. A `build/Release/generators/` directory (and similarly for Debug) containing files like `conan_toolchain.cmake` and `physkit-release-conan-presets.json` (the name may vary) which includes the Conan CMake presets.
3. A `CMakeUserPresets.json` in the project root. This user presets file is auto-generated by Conan to include the presets from the above generator folders[19]. It basically tells CMake how to use the Conan toolchain and where the build directory is for each preset.

If the `CMakeUserPresets.json` was not created, it could be that your CMake version is below 3.23 or presets were disabled. In that case, you can manually set up the presets (covered in the next section) or use the classic CMake invocation. For example, as a fallback you could run CMake manually:

```
cmake -B build/Release -S . -G Ninja -
DCMAKE_TOOLCHAIN_FILE=build/Release/generators/conan_toolchain.cmake -
DCMAKE_BUILD_TYPE=Release
```

(and similarly for Debug). However, using the generated presets is easier if available.

1. **Adding new dependencies:** To add another library via Conan, you would edit `conanfile.py` and add a `self.requires("package/version")` line in the `requirements()` method (similar to the mp-units line[15]). Then run the `conan install` commands again. Conan will fetch the new dependency and generate updated configs. Always check ConanCenter for the exact name/version of a package and whether options are needed. If Conan fails to find a package, ensure the remote (ConanCenter) is set up (it is by default) and the package name is correct.

Configuring CMake Presets for the Project

Conan 2.x has seamlessly integrated with CMake Presets to simplify the configuration step. After the above `conan install` steps, you should have a **CMakeUserPresets.json** ready to use. This file includes references to Conan-generated presets that already specify the toolchain file and other necessary cache variables for CMake[18].

Let's break down what's been set up and how to use it:

- **Understanding the Presets:** Open `CMakeUserPresets.json` in a text editor. You will see that it uses CMake's `"include"` mechanism to include one or more Conan-generated `CMakePresets.json` files from the `build/<config>/generators` directories[19]. Each of those contains a configure preset (and associated build/test presets) named with a `conan-` prefix. For example, you might see presets like `"conan-release"` and `"conan-debug"` for single-config generators, or a `"conan-default"` (configure) plus `"conan-release"`/`"conan-debug"` (build presets) for multi-config. These presets encode:
- The path to the Conan toolchain file (via `CMAKE_TOOLCHAIN_FILE`).
- The build directory (`binaryDir`) for that configuration (e.g., `build/Release`).
- The generator being used. By default, Conan may choose the platform's default CMake generator (on Windows, likely Visual Studio; on macOS, Unix Makefiles). You can influence this by setting `tools.cmake.cmaketoolchain:generator=Ninja` in your **Conan config** or by specifying a generator in `cmake_layout(self, generator)` in the recipe. We recommend using **Ninja** for both macOS and Windows for consistency (and faster builds), but you can use Visual Studio's generator on Windows if you prefer an MSVC solution file.

- **Using Ninja vs. Visual Studio Generators:** For VSCode and clangd integration, Ninja is preferable because it produces a `compile_commands.json` (Visual Studio's generator does not support compile commands)[20]. If your Conan presets defaulted to Visual Studio on Windows, you have two options:

- Re-run `conan install` with a config option to use Ninja (e.g., `conan install . -s build_type=Release -c tools.cmake.cmaketoolchain:generator=Ninja ...`), or
- Manually edit the CMakeUserPresets.json to change the generator to "Ninja" for the included presets (be careful: if you do this, also adjust any VS-specific fields like toolset if present).
  For macOS, the default "Unix Makefiles" can be used, but Ninja (installable via brew) is typically faster and also yields compile_commands. You can similarly adjust to Ninja if desired.

- **Customizing or Creating Presets:** If Conan's auto-generated presets meet your needs, you don't need to create your own `CMakePresets.json`. However, you **can** add your own presets or overrides in CMakeUserPresets.json if you have special requirements. According to CMake convention, `CMakePresets.json` (checked into

VCS) is for project-wide presets, and `CMakeUserPresets.json` is for each developer's local customizations[21]. In our case, Conan wrote the user presets file for us. It's fine to use it directly. If you wanted, for example, to add `CMAKE_EXPORT_COMPILE_COMMANDS` or change the compiler, you could duplicate a preset entry in CMakeUserPresets and tweak it:

```
{
  "version": 4,
  "configurePresets": [
    {
      "name": "my-release",
      "inherits": "conan-release",
      "cacheVariables": {
        "CMAKE_EXPORT_COMPILE_COMMANDS": "ON",
        "CMAKE_C_COMPILER": "/usr/local/opt/llvm/bin/clang",
        "CMAKE_CXX_COMPILER": "/usr/local/opt/llvm/bin/clang++"
      }
    }
  ]
}
```

This example inherits Conan's release preset but turns on compile commands and forces a specific compiler path (useful on macOS to ensure Homebrew clang is used). In practice, simply setting environment variables or using the Conan profile correctly should already point CMake to the right compiler, as long as you launch CMake in the proper environment (e.g., Developer Command Prompt for MSVC, or with CC/CXX set for custom compilers). Use this advanced approach if needed.

- **Ensure CMake version**: Make sure you have CMake **3.23 or newer**. Presets (especially the include mechanism used by Conan) require CMake 3.23+[22]. You can check with `cmake --version`. If using VS 2022 or latest CMake from Homebrew, you should be fine.

Building the Project with CMake

With dependencies installed and presets configured, building PhysKit is straightforward:

- **Using CMake from the command line:** You can invoke CMake with the generated presets:
- Configure step (if using a multi-config generator like Visual Studio on Windows):

  ```
  cmake --preset conan-default
  ```

  This will configure the CMake project (generating the Visual Studio solution or build files). Conan's preset ensures the toolchain and dependencies are set up. For a multi-config, you do this once.

- Build step:
  For multi-config (VS generator), build using the build presets for the desired configuration, e.g.:

  ```
  cmake --build --preset conan-release
  ```

  This will invoke MSBuild or Ninja to compile in Release mode using the solution/build files from the configure step[23]. Similarly, you can build the Debug preset (`cmake --build --preset conan-debug`). For single-config generators (Makefiles/Ninja), you can often combine configure and build by using the same preset name. For example, if `conan-release` is a configure preset for Ninja, you might do:

  ```
  cmake --preset conan-release
  cmake --build --preset conan-release
  ```

  (In CMake >=3.20, a build preset can be auto-generated or implicitly used. Conan may have created matching build presets for Ninja as well, making this seamless[24].)

- After building, an executable or library should be produced according to the project's CMakeLists. You can also run CTest if tests are defined. Conan's recipe calls `cmake.ctest()` if `can_run(self)` in the Conan local cache build, but when building locally you can manually run tests. If a `test` preset exists (e.g., `conan-release-test`), you can do `cmake --build --preset conan-release --target test` or use `ctest` in the build directory.

- **Using VSCode with CMake Tools (Recommended):** The next section covers this in detail, but essentially VSCode will detect these presets and provide a GUI to configure and build without manual CLI commands.

Troubleshooting Build Issues: - If CMake fails to configure, read the error closely. Common issues include **mismatched compiler** (e.g., Conan profile says GCC but CMake found MSVC – make sure you launched CMake with the intended compiler environment as noted earlier[12]) or **unsupported flags** (e.g., Apple Clang rejecting `-std=c++26`. In such a case, double-check you're using the Homebrew clang. You might see an error like *unrecognized command-line option '-std=c++26'* with Apple Clang. The solution is to use a newer compiler as discussed or use the `-std=c++2c` flag if absolutely stuck on that compiler, but generally using the proper compiler is preferred).

- If Conan couldn't find a binary for your profile and failed building a dependency: possibly your compiler is very new. For example, if you set `compiler.version=21` for clang (LLVM 21, hypothetical) and Conan's package recipes don't know it, you might need to update your Conan settings or use the nearest version (20) in the

profile. Alternatively, upgrade Conan; newer Conan releases update the default `settings.yml` to include newer compiler versions. You can also manually edit `~/.conan2/settings.yml` to add an entry (noting this is advanced and not usually needed if Conan is up-to-date).

- If mp-units fails to build with modules enabled: C++20 modules are still experimental. Ensure you have a compiler that supports modules (Clang 15+ or MSVC 19.3+). If issues persist, you could try disabling modules by changing the Conan option (`cxx_modules=False`) in conanfile or using a different version of mp-units. However, since the project specifically enabled modules, the intention is to use them. Modern Clang tends to handle modules better than GCC or even MSVC, so using LLVM Clang is a good choice here.

Integrating with VSCode (CMake Tools Extension & Conan)

Using VSCode can greatly streamline the development workflow. We will integrate Conan/CMake through the **CMake Tools** extension, which has direct support for CMakePresets.

Setup VSCode and extensions: - Install the **CMake Tools** extension (if not already). Also install the **clangd** extension for C++ code navigation (and disable the MS C++ extension if you prefer clangd). - Open the PhysKit project folder in VSCode. The extension should detect the presence of `CMakePresets.json`/`CMakeUserPresets.json` and automatically switch to using presets (you can confirm in VSCode's settings that `"cmake.useCMakePresets": "auto"` or `"always"` is enabled).

Using CMake Tools with presets: - **Select Configure Preset:** In the VSCode status bar or command palette (`Ctrl+Shift+P` > **CMake: Select Configure Preset**), choose one of the available presets. You should see options corresponding to the Conan-generated presets, e.g., **conan-release** and **conan-debug** (for Ninja) or **conan-default** (for Visual Studio). Select the one you want to work with (e.g., conan-debug for development). - Once selected, trigger **Configure** (VSCode may do this automatically upon selection). This will run the `cmake --preset ...` behind the scenes. Check the **CMake/Build** output in VSCode to ensure it finished without errors. - **Select Build Preset / Build Variant:** If you selected a configure preset that is single-config (like conan-debug for Ninja), VSCode knows the build type. If using a multi-config, you might need to also pick a build preset or at least select the active configuration (Debug/Release) in the status bar. CMake Tools will show a "Build" button (▶️) – click it to compile. It will invoke the appropriate `cmake --build` command for the chosen preset. - **Debug/Run:** If an executable is produced, you can create a launch configuration in VSCode to run or debug it. CMake Tools can generate a default debug configuration for you if it recognizes the target. Otherwise, use the Run tab to set one up (point it at the built executable under `build/Debug` or `build/Release`). Running tests can be done via CTest or the CMake Tools "CTest" sidebar if enabled.

Conan integration in VSCode: Since we ran `conan install` manually, the dependencies are already set. Alternatively, some workflows integrate Conan into VSCode tasks or CMakeLists (e.g., using the `cmake-conan` helper) to auto-run Conan on configure. In our case, the presets approach means you only need to rerun the Conan install step when dependencies change or when you first set up the project. Day-to-day building does not require re-running Conan unless you modify `conanfile.py` or want to update dependencies.

Intellisense and autocompletion: With CMake Tools configured, VSCode will use the information from the active kit/preset to provide C++ IntelliSense. If using the clangd extension, it will read compile commands for better accuracy (discussed next). The CMake Tools extension itself also configures the include paths/defines for the MS C++ extension if you were using that, but we recommend clangd for standard compliance.

Generating and Using compile_commands.json for Clangd

The **clangd** language server relies on a `compile_commands.json` file (compilation database) to understand how to compile the project (include paths, flags, etc.)[25]. We need to generate this file from CMake and ensure clangd can find it:

1. **Enable compile commands in CMake:** We should ensure that CMake is exporting `compile_commands.json`. This is done by setting the cache variable `CMAKE_EXPORT_COMPILE_COMMANDS` to ON. If you used the Conan presets out-of-the-box, this might not be enabled by default. You have a few ways to enable it:
2. Easiest: add the setting in a `CMakeUserPresets.json` entry as shown earlier (under `cacheVariables`). For instance, you could add:

   ```
   "cacheVariables": {
       "CMAKE_EXPORT_COMPILE_COMMANDS": "ON"
   }
   ```

   to each configure preset (or to a parent preset that others inherit).
3. Or manually run CMake with `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` once. For example:

   ```
   cmake --preset conan-debug -D CMAKE_EXPORT_COMPILE_COMMANDS=ON
   ```

   (You might need to delete your CMakeCache.txt or use a fresh build folder if re-configuring with new flags, but since we can edit presets, that's cleaner.)
4. If you edit the preset JSON, reloading VSCode or re-configuring will apply it. After enabling this, CMake will generate a `compile_commands.json` in the build directory for that preset every time you configure.

5. **Locate the compile_commands.json:** After a successful configure, check `build/Debug/compile_commands.json` (and likewise for Release if configured). This JSON lists all source files and the exact compile command used for each.

6. **Make clangd use it:** clangd will automatically search for a compile_commands.json in parent directories of source files or in a folder named `build`[26]. If your build directory is the default `build/` under the project root, clangd will find it with no extra effort. For example, editing a file in `src/` will cause clangd to look in `../build/` and it should locate the database[26]. If you used a different build directory name or location, clangd might not find it. In that case, you have two options:

7. **Option A:** Symlink or copy the compile_commands.json to the project root. The clangd docs note that if your build directory isn't `$PROJECT/build` or is named differently, you can simply do:

   ```
   ln -s build/Debug/compile_commands.json compile_commands.json
   ```

   in the root (or copy the file)[27]. Clangd will then see it immediately at the root.
8. **Option B:** Configure clangd's path. In VSCode settings, you can set `"clangd.arguments": ["--compile-commands-dir=build/Debug"]` (or whichever build folder you want clangd to use). This flag tells clangd exactly where to look. This is useful if you have multiple build directories or if your compile_commands is not in a standard location.

For simplicity, if you primarily work in one configuration (say Debug), you could copy that compile_commands.json to the root whenever it changes. Alternatively, open the `build/Debug` folder as part of your workspace in VSCode (clangd will then find it as a subdirectory named "build").

1. **Verify clangd is working:** In VSCode, open a C++ source file. If clangd is properly using the compile commands, you should see that it recognizes includes (no squiggly lines for included headers that are installed via Conan, for example) and provides autocompletion that matches your project and its dependencies. If you still see include errors, double-check that compile_commands.json is up to date and that clangd is pointing to it. Remember that Visual Studio's generator will **not** produce a compile_commands.json[20], so if you originally configured with VS generator, switch to Ninja for clangd's sake.

Troubleshooting & FAQ

Q: Conan says a setting is not set or not known (e.g., compiler.cppstd) – This means your profile might be missing that line, or you're using a value not recognized. Edit the profile to add `compiler.cppstd` as described. Use `conan profile show [profilename]` to review the profile's contents. If using a very new standard or compiler, make sure Conan's settings.yml supports it – update Conan if necessary.

Q: Conan couldn't find a pre-built package for my compiler – Conan will then try to build from source. If that fails (perhaps due to your compiler's quirks), you might need to adjust options. For instance, mp-units with modules may fail on GCC; using Clang or MSVC might

be necessary. You can also try disabling modules (set `cxx_modules=False` in conanfile) as a last resort if the compiler cannot handle them.

Q: CMake configure fails with "incorrect compiler, is not the one detected by CMake" – This happens if, for example, you set `compiler=gcc` in profile but CMake is actually using Clang, etc. The solution is to ensure environment consistency: run `conan install` and `cmake` with the same default compiler. If you want to use a non-default compiler, set the `CMAKE_C_COMPILER`/`CMAKE_CXX_COMPILER` in the preset or environment. Remember, **Conan does not choose the compiler for you; it assumes you will use the one you told it**[12].

Q: How do I add a new library dependency? – Add it to `conanfile.py` (in `requirements()`) and possibly in CMakeLists (find_package or target_link_libraries if not using Conan's auto-generated config files). Then run `conan install` again. Commit the updated conanfile so others can run it too. If the library is not header-only, you may need to adjust your CMake to link to the Conan package's targets. Conan's CMakeDeps will generate find modules or config files for each dependency (e.g., `find_package(mp-units CONFIG)` should work after running Conan, because CMakeDeps will generate `mp-units-config.cmake` pointing to the package[17]).

Q: Should I commit the CMakeUserPresets.json? – Generally, **no**, you shouldn't commit the user presets file, as it's intended for user-specific settings (and Conan generates it locally). You can, however, commit a `CMakePresets.json` with some baseline presets if you want everyone to have them. But since Conan handles it here, most of the heavy lifting is done. Each developer can just run the Conan commands and get their own user preset file.

Q: Apple Clang issues on macOS – Make sure you installed a modern Clang via Homebrew (`brew install llvm`). If you get link errors about system libraries (like a missing `-lSystem` as in some forum discussions), it might be because the Homebrew Clang isn't finding the macOS SDK. Using Xcode's clang for linking or specifying `-isysroot` correctly can resolve that. A quick fix is to tell CMake to use the Apple SDK with the new compiler by setting `SDKROOT` environment or `CMAKE_OSX_SYSROOT`. Typically, if Xcode Command Line Tools are installed, CMake should fill this in. The majority of cases won't run into this, but it's something to note if you see linker errors about missing macOS frameworks when using non-Apple clang[28][29]. The advice from experienced macOS developers is that mixing Homebrew LLVM with Xcode tools can be tricky; using an Xcode-provided toolchain for linking and Homebrew clang for compiling is one approach, or use a VM, etc., but hopefully you won't need to go that far for PhysKit.

Q: Visual Studio/VSCode doesn't let me select Ninja – Newer Visual Studio versions might default to their own CMake generator. If using the MSVC IDE, you can enable Ninja by installing it and selecting it in CMake Settings or Presets. In VSCode, just ensure the preset specifies Ninja. If VSCode's CMake Tools still opens the MSVC solution, double-check that the preset is actually being used (the status bar should show `[Preset]` not [Kit]). You can force `"cmake.useCMakePresets": "always"` in your settings to avoid Kits.

By following this guide, you should have a functional development setup for PhysKit on Windows or macOS, using Conan to manage dependencies, CMake Presets for configuration, and VSCode for an IDE. Your builds will use modern C++ compilers (LLVM/Clang 16+ or GCC 12+/MSVC 2022) to support C++26 features, and clangd will provide rich code insights thanks to the compile commands database. Happy coding!

Sources:

- Conan installation methods (pip, brew)[1][3]
- Conan profile detection and default cppstd[8][9]
- Note on sharing profiles in teams[14]
- Apple Clang vs. modern LLVM (lagging C++ features)[7]
- MSVC 2022 supporting C++23/26 features (/std:c++latest)[11]
- Conan's CMake integration (CMakeToolchain, presets)[17][18]
- CMakePresets vs UserPresets usage in teams[21]
- Ninja generator needed for compile_commands (VS generator doesn't support it)[20]
- Conan profile vs actual compiler – require setting CMake compiler explicitly[12][13]
- Enabling and using compile_commands with clangd (search paths and symlink advice)[26][27]

---

[1] [2] [3] [4] [5] [6] Install — conan 1.66.0 documentation

https://docs.conan.io/1/installation.html

[7] [28] [29] Anybody using llvm 19 (or not-AppleClang) on macOS? - MacOSX and iOS - JUCE

https://forum.juce.com/t/anybody-using-llvm-19-or-not-appleclang-on-macos/64885

[8] [9] [14] conan profile — conan 2.20.1 documentation

https://docs.conan.io/2/reference/commands/profile.html

[10] C++ Programming Language Status - Clang

https://clang.llvm.org/cxx_status.html

[11] What's new for C++ in Visual Studio | Microsoft Learn

https://learn.microsoft.com/en-us/cpp/overview/what-s-new-for-visual-cpp-in-visual-studio?view=msvc-170

[12] [13] Conan profile: you tell Conan what your compiler is, not the other way around – Ivan Krivyakov

https://ikriv.com/blog/?p=4879

[15] [16] conanfile.py

https://github.com/jcsq6/PhysKit/blob/8587c2b651093617e09e55f6450f5eb55b2e0871/conanfile.py

[17] [18] [19] [22] [23] [24] CMakeToolchain: Building your project using CMakePresets — conan 2.20.1 documentation

https://docs.conan.io/2/examples/tools/cmake/cmake_toolchain/build_project_cmake_presets.html

[20] Sonarlint VS 2022 with CMake Presets - Visual Studio - Sonar Community

https://community.sonarsource.com/t/sonarlint-vs-2022-with-cmake-presets/98097

[21] CMake Presets integration in Visual Studio and Visual Studio Code - C++ Team Blog

https://devblogs.microsoft.com/cppblog/cmake-presets-integration-in-visual-studio-and-visual-studio-code/

[25] [26] [27] Getting started

https://clangd.llvm.org/installation