# Analysis of User Experience and Best Practices for HPC Build Tools in the E4S 25.06 Ecosystem

## 1. Executive Summary

This report provides an expert-level analysis of the build tools within the Extreme-scale Scientific Software Stack (E4S) 25.06 release, based on user experiences and technical documentation. The analysis focuses on Spack, CMake, and GNU Autotools, with a specific mandate to characterize user preferences and provide guidance on tool selection based on programming language, project complexity, and the management of GPU dependencies (CUDA, HIP, SYCL).

The central finding is that these tools are not competitors but components of a sophisticated, multi-layer build hierarchy. Spack, as the high-level package manager (Layer 3), is the core integration technology of E4S, orchestrating the execution of meta-build systems (Layer 2) like CMake and Autotools.1 Spack’s primary user experience is its powerful "spec" syntax, which allows scientists and administrators to manage the combinatorial complexity of compilers, libraries, and build variants (e.g., +cuda, %gcc, cuda\_arch=80).3

For the meta-build system (Layer 2), CMake is the undisputed de facto standard for modern HPC applications. Its user experience is vastly superior for the two most critical scientific computing use cases: mixed-language C++/Fortran projects, where it automates complex linking and module dependencies 6, and GPU-centric development. CMake provides "first-class," native language support for both CUDA and HIP via the enable\_language() command, a simple and robust user experience.7

Conversely, GNU Autotools is identified as a legacy tool whose user experience with modern GPU compilers is demonstrably poor and, in some cases, fundamentally broken. Forum discussions and technical reports document a history of user failures, from simple linker and flag conflicts 9 to systemic incompatibilities with the latest NVIDIA HPC SDK compilers.11

The optimal stack for E4S users is the symbiotic combination of Spack and CMake. Spack automates the most difficult aspect of CMake (dependency management) 12, while CMake provides a robust, modern, and GPU-aware build logic for the application itself. Finally, new tools in the E4S 25.06 ecosystem, such as e4s-alc for custom container generation 14 and e4s-cl for high-performance MPI container launching 15, complete the full "build-to-deployment" lifecycle for HPC users.

This report recommends standardizing on CMake for all new C++, Fortran, and GPU-based application development, leveraging Spack for integration and deployment, and explicitly forbidding the use of Autotools for any new GPU-centric projects.

## 2. Deconstructing the HPC Build Ecosystem: Package Manager vs. Build System

A frequent point of confusion among scientific software users is the distinction between a package manager like Spack and a build system like CMake. The E4S ecosystem clarifies this by organizing these tools into a distinct hierarchy.2 Understanding this hierarchy is the first step in selecting the correct tool.

### 2.1. The Hierarchical Model

The E4S build process operates on three primary layers:

1. **Low-Level Build System (Layer 1):** This layer consists of tools like Make and Ninja. Their job is to execute commands and manage dependencies among individual files within a single project. Users and developers rarely, if ever, interact with this layer directly.2
2. **Meta-Build System (Layer 2):** This layer includes **CMake** and **GNU Autotools**. They provide a higher-level abstraction for developers, who write configuration files (e.g., CMakeLists.txt or configure.ac). These tools then *generate* the Layer 1 Makefiles appropriate for the platform.2 Their domain is the build logic *inside* a single software package.
3. **Package Manager (Layer 3):** This layer is occupied by **Spack**. Its domain is *not* the internals of one package, but the complex dependency graph *between* all packages in a scientific software stack.1 It is the "General Contractor" that manages the overall environment and directs the execution of the "Subcontractors" (CMake/Autotools) at Layer 2.2

### 2.2. Spack: The HPC Package Manager (Layer 3)

Spack's role is to automate the installation and management of the entire software stack.18 It is the foundational technology for E4S.20 Its purpose is to manage the combinatorial complexity of HPC software, where a single application may have dozens of dependencies that, in turn, require specific compilers, MPI implementations, or library versions.22 Spack solves this by managing dependencies, driving the package-level build systems, and ensuring consistent, reproducible builds.1

### 2.3. CMake and Autotools: The Meta-Build Systems (Layer 2)

CMake and Autotools operate at a lower level of abstraction. Their role is to handle library abstractions and generate the Makefiles or Ninja files used for compilation.2 Spack does not replace these tools; rather, packages built by Spack can use *any* build system they want.1 For example, the Spack recipe for a package will invoke cmake or ./configure with the correct flags to build that specific package.16 The user experience for a developer writing a new application is therefore focused on Layer 2 (writing a CMakeLists.txt), while the user experience for a scientist *using* that application is focused on Layer 3 (running spack install).

## 3. Spack: The Core of E4S 25.06 Software Delivery

As the software deployment tool for the Exascale Computing Project (ECP), Spack is the most depended-upon project in the E4S ecosystem.16 Its user experience is tailored to two distinct groups: the end-user (scientist or facility administrator) and the packager (application developer or integration team).

### 3.1. User Experience for Scientists and HPC Administrators

For end-users, Spack's primary interface is its powerful "spec" (specification) syntax.3 This domain-specific language allows a user to concisely request highly specific software configurations from the command line. This resolves the HPC problem where "one size fits all" is infeasible.22

Key user experiences of the spec syntax include:

* **Version Selection:** spack install hdf5@1.10.5 2
* **Compiler Selection:** spack install mpileaks %gcc@4.7.3 5
* **Build Options (Variants):** spack install hdf5 +hl +threadssafe 2
* **Microarchitecture Targeting:** spack install hdf5 target=cascadelake 2
* **Dependency Customization:** spack install mpileaks ^mpich@3.2 2
* **GPU Variants:** spack install hipsycl +cuda +rocm 24

Spack is language-agnostic, easily handling stacks with C, C++, Fortran, Python, and R packages.25 All installations are non-destructive and "peacefully coexist," as each unique configuration is installed to its own hash-based prefix.3

For HPC administrators, such as those at NERSC, the user experience involves defining the site's software stack using Spack's environment files (e.g., spack.yaml).26 Administrators take the E4S reference spack.yaml, customize it to point to site-provided compilers and "external" packages, and then use Spack to build the stack and generate the corresponding module files for users.26 This workflow is highly automated and reproducible. The E4S 25.06 release further enhances this by providing binary build caches, allowing users to install pre-compiled packages in seconds rather than hours.21

### 3.2. The Spack package.py File: The Packager's Experience

For the developer or packager, the experience is defined by the package.py recipe file. This is a single, pure Python file that defines *all* possible build configurations for a given software package.2 This is a major advancement over older systems that required a profusion of files for different configurations.22

A typical package.py file defines:

* **Variants:** Conditional build options, like variant('mpi', default=True, description='Build with MPI.').5
* **Dependencies:** Conditional dependencies, like depends\_on('mpi', when='+mpi').5
* **Build Logic:** It inherits from a base class (e.g., CMakePackage, AutotoolsPackage) that automates 80% of the build.30 The packager just overrides functions to pass in specific arguments, such as def cmake\_args(self): return.5

This model provides a clean, standardized, and highly expressive Python DSL for managing what would otherwise be a cache of "magic configure lines".1

### 3.3. The Spack + CMake Synergy

The user experience for C++ developers, in particular, is defined by the symbiotic relationship between Spack and CMake. A 2023 ISO C++ developer survey found that dependency management is the number one frustration for over 80% of C++ developers.13 This is precisely the problem Spack solves for CMake.

This synergistic workflow proceeds as follows:

1. **Developer (Layer 2):** An application developer writes a CMakeLists.txt file. To find a dependency like HDF5, they write find\_package(hdf5 REQUIRED).
2. **CMake (Layer 2):** CMake's find\_package() command works by searching a list of directories specified in the CMAKE\_PREFIX\_PATH variable.12
3. **End-User (Layer 3):** A scientist runs spack install my-app.
4. **Spack (Layer 3):** Spack first identifies and builds all dependencies of my-app, including HDF5.
5. **Integration (Layer 3 -> 2):** When Spack invokes CMake to build my-app, its CMakePackage base class *automatically* populates the CMAKE\_PREFIX\_PATH variable with the exact installation prefixes for *all* transitive dependencies of my-app.12

The result is that the developer's find\_package(hdf5) call "just works," with no manual path management required from the user or developer. Spack (Layer 3) automates the most difficult and frustrating part of the CMake (Layer 2) user experience.

## 4. CMake: The De Facto Standard for HPC Application Development

Within Layer 2 (meta-build systems), CMake is the clear de facto standard for C++ code 36 and the superior choice for modern scientific computing, particularly for mixed-language and GPU-based projects.

### 4.1. User Experience: C++, Fortran, and Mixed-Language Projects

For C++ developers, "Modern CMake" (versions 3.15+) provides a clean, powerful, and target-based experience that avoids the "anti-patterns" of older, global-variable-based CMake.37

For large-scale Fortran and mixed C++/Fortran scientific codes, CMake's user experience is a best-in-class, "killer feature".6 It automates the most painful aspects of mixed-language builds:

* **Automatic Build Order:** CMake contains a full Fortran parser. It automatically determines which Fortran files produce (.mod) modules and which consume them, ensuring the correct compilation order. This feature requires no user interaction.6
* **Automatic Runtime Linking:** When a C/C++ executable links to a Fortran library, CMake automatically detects the required Fortran runtime library and adds the correct linker flags.6 This automates a process of "iterate[d] over until the build works" that developers previously did by hand.6
* **Automatic Symbol Mangling:** CMake's FortranCInterface module automatically determines the compiler's name-mangling scheme (e.g., adding underscores) and generates a C/C++ header file with macros to correctly call Fortran from C/C++ and vice-versa.6

The developer user experience is simplified to just listing Fortran and C or CXX in the project() command.6

### 4.2. User Experience: Managing GPU Platform Portability

CMake is the only meta-build system with robust, "first-class" support for GPU compilers, which is critical for E4S.42

**CUDA (NVIDIA):**

* **The Old Way (Bad UX):** Projects used to rely on a FindCUDA.cmake module.43 This module is now **deprecated** and should not be used in new code.7
* **The Modern Way (Good UX):** Since CMake 3.8, CUDA is a "first-class" language. The user experience is simply adding CUDA to the project() command or calling enable\_language(CUDA).7 CUDA (.cu) files can then be added directly to add\_executable() or add\_library() targets just like C++ files.7

**HIP (AMD):**

* **The Modern Way (Good UX):** Following CUDA's example, CMake 3.21 added first-class language support for HIP.8 The user experience is identical: enable\_language(HIP).

Portability Strategies:

For projects that must support both NVIDIA (CUDA) and AMD (HIP) GPUs, developers use CMake to manage portability at two levels.

1. **Build System Portability:** The CMakeLists.txt file uses a variable (e.g., GPU\_RUNTIME) to switch logic. A common user experience is:  
   CMake  
   # User can set -DGPU\_RUNTIME=CUDA or -DGPU\_RUNTIME=HIP  
   if(GPU\_RUNTIME STREQUAL "HIP")  
    target\_link\_libraries(my\_app PRIVATE amdhip64)  
   elseif(GPU\_RUNTIME STREQUAL "CUDA")  
    target\_link\_libraries(my\_app PRIVATE cudart)  
   endif()  
     
   This pattern allows a single CMakeLists.txt file to target both backends.49
2. **Code-Level Portability:** The build system logic is paired with C preprocessor macros in the code. A common common.h header file will abstract the API:  
   C++  
   #**ifdef** HIP\_ENABLED  
   #**include** <hip/hip\_runtime.h>  
   #**define** devError\_t hipError\_t  
   #**define** devEventCreate hipEventCreate  
   #**else**  
   #**include** <cuda\_runtime.h>  
   #**define** devError\_t cudaError\_t  
   #**define** devEventCreate cudaEventCreate  
   #**endif**  
     
   The C++ application code then uses the generic devError\_t type. The CMakeLists.txt is responsible for passing the -DHIP\_ENABLED define when GPU\_RUNTIME is set to HIP.50 This provides a clean and portable developer experience.

SYCL (Intel/Multi-platform):

SYCL is a Khronos standard, not a specific compiler.51 SYCL implementations, like hipSYCL 53 or Intel DPC++ 52, are themselves CMake-based projects. The user experience for a SYCL application developer is to use CMake to find the installed SYCL implementation, which then transparently handles its own backends (e.g., CUDA, HIP, or OpenMP).54

## 5. GNU Autotools: The Legacy Portability Workhorse

The GNU Autotools suite is a key part of the E4S stack, as it is the build system for many foundational, legacy HPC packages.

### 5.1. User Experience and Strengths

The Autotools suite is comprised of three main tools:

* **Autoconf:** Reads configure.ac (written in m4 macros) to generate the ./configure shell script.56
* **Automake:** Reads Makefile.am to generate a portable Makefile.in template.56
* **Libtool:** Helps create and use shared libraries portably.56

**End-User Experience (Pro):** The primary strength of Autotools is the simple, universal end-user experience: ./configure && make && make install.64

**Developer Experience (Pro):** The configure script's "superpower" is its portability across "a LOT of broken toolchains".66 Instead of *assuming* a feature exists (like CMake often does), the shell script *probes* the host system to test for everything it needs.60 This makes it extremely robust on esoteric or older UNIX-like systems.

**Developer Experience (Con):** The developer experience is widely considered "overly complicated".67 It requires learning the m4 macro language, which users describe as "disgusting m4 scripts".37 Furthermore, it has no viable support for Windows targets.66

### 5.2. Analysis of Weaknesses: Documented User Failures with GPU Toolchains

For the E4S 25.06 ecosystem, the most significant finding is that the Autotools user experience is demonstrably poor to non-functional for modern GPU development. This is not a single bug but a systemic, long-term pattern of failure.

* **2011 User Failure (libtool vs nvcc):** A user on the NVIDIA forums reports that libtool (the Autotools shared library handler) incorrectly passes the -fPIC flag to nvcc (the NVIDIA CUDA compiler), which does not understand it. The user must manually write complex Makefile.am rules to pass the correct flag, -Xcompiler "-fPIC", to nvcc. This is a brittle, non-obvious workaround for a broken user experience.9
* **2015 User Failure (Linking):** A user reports being "crazy" trying to modify an existing Autotools-generated Makefile to integrate .c and .cu files. They are unable to resolve "undefined reference error" issues 10, a common linker problem when C and CUDA code are not handled correctly by the build system. This highlights the manual effort required for a task that CMake automates.
* **2024 User Failure (Systemic Incompatibility):** The most critical evidence is a 2024 report that the latest NVIDIA HPC SDK (v23.11) "does not work with Autoconf/Automake-based projects".11 The failure is fundamental:
  1. The autoconf-generated configure script checks compiler features by running nvc -E (the C preprocessor).
  2. The nvc compiler, unlike gcc, outputs C++ struct definitions during its pre-include phase.
  3. The configure script, being a shell script, *captures this C++ code* and saves it into a *shell variable*.
  4. The subsequent make command fails with a "syntax error near unexpected token }" because the shell attempts to interpret the C++ struct syntax stored in the variable.11

This 2024 failure demonstrates a fundamental design incompatibility. Autotools is a GNU-centric system that *assumes* compilers will behave like gcc. Modern, non-GNU compilers like nvc violate this assumption, breaking the Autotools build process itself. This makes Autotools a clear anti-pattern for new GPU-based software.

## 6. E4S 25.06 Auxiliary Build and Deployment Tools

The E4S 25.06 release is not just Spack and its packages; it includes a suite of auxiliary tools that complete the build-to-deployment lifecycle, primarily for containerized workflows.

### 6.1. e4s-alc: User Experience for "À la Carte" Custom Container Builds

* **Tool:** e4s-alc ("E4S à la carte").14
* **Problem Solved:** Full E4S container images can be very large (e.g., 25GB 72), and users often need minimal, application-specific images.
* **User Experience:** e4s-alc provides a simple user experience for building "custom compact container images".73 The user provides a simple .yaml file as input.14 This file specifies:
  1. A base image (e.g., ecpe4s/e4s-base-cuda:24.05 71).
  2. A list of OS packages (e.g., libgomp).70
  3. A list of **Spack packages** to install.70
* e4s-alc then reads this file and automatically generates the complex Dockerfile or Singularity definition file, "eliminat[ing] manual definition files scripting".14

### 6.2. e4s-cl: The Role of MPI Substitution in the Deployment Lifecycle

* **Tool:** e4s-cl ("E4S Container Launch").15
* **Problem Solved:** An MPI application inside a container is built against the generic MPI *within* that container. To achieve performance on a supercomputer, that application *must* use the high-performance, vendor-tuned MPI on the *host* system.73
* **User Experience:** e4s-cl is a launch wrapper that "will run the MPI binary in the target container using the host MPI library".15 It performs "MPI library substitution" 29 for any ABI-compatible library (e.g., MPICH, Intel MPI, Cray MPICH).15 The user's command is simplified to: e4s-cl mpirun -np 4./a.out.29

These tools, when combined, create a complete and coherent workflow:

1. **Build:** Spack builds the individual software packages.79
2. **Package:** e4s-alc uses Spack to assemble those packages into a minimal, portable container.73
3. **Deploy:** e4s-cl runs that container, substituting the host's high-performance MPI to achieve native performance.29

## 7. Comparative Analysis and Strategic Recommendations

The E4S 25.06 ecosystem provides clear evidence to guide tool selection. The choice is not *between* Spack, CMake, and Autotools, but about selecting the right Layer 2 (meta-build) tool to be orchestrated by Spack (Layer 3).

**Table 1: Comparative Analysis of HPC Build Tools in E4S**

| **Feature** | **Spack (Package Manager - Layer 3)** | **CMake (Meta-Build System - Layer 2)** | **GNU Autotools (Meta-Build System - Layer 2)** |
| --- | --- | --- | --- |
| **Primary Role** | Manages installation, configuration, and dependencies of the *entire* software stack.1 | Generates build files (e.g., Makefiles) *within* a single software package.2 | Generates a Makefile *within* a single software package.56 |
| **Primary User** | **End-User/Scientist:** Installs software (spack install...).2 **Admin/Integrator:** Deploys site-wide software.26 | **Package Developer:** Writes CMakeLists.txt to define their project's build logic.38 | **Package Developer:** Writes configure.ac and Makefile.am.62 |
| **C++ Support** | Excellent. Manages C++ dependencies, solving the #1 C++ developer frustration.13 | **Excellent.** The de facto standard.36 "Modern CMake" is powerful and target-based.38 | **Viable.** Supported 57, but less common for new projects. Inferior to CMake. |
| **Fortran & Mixed-Language** | Excellent. Manages Fortran compilers and libraries.23 | **Excellent (Best-in-Class).** Automatically handles module order 6, runtime linking 6, and name-mangling.6 | **Viable but Difficult.** Manually linking Fortran/C is the "old way" that CMake automates.6 |
| **GPU (CUDA/HIP/SYCL) Support** | **Excellent.** Provides a unified user-facing variant system (+cuda, cuda\_arch=..., +rocm) to abstract and control GPU builds.2 | **Excellent (Best-in-Class).** Provides "first-class" native language support via enable\_language(CUDA) 7 and enable\_language(HIP).8 | **Extremely Poor (Anti-Pattern).** Documented, systemic user failures spanning a decade, from libtool conflicts 9 to fundamental incompatibility with NVIDIA HPC SDK.11 |
| **Key User Experience (Pro)** | Simple, powerful spec syntax.3 E4S build caches.27 Reproducible environments.4 | "Modern CMake" is clean and powerful.38 Unmatched C++/Fortran support.6 Native GPU language support.7 | The simple, classic ./configure && make && make install for end-users.64 Excellent portability on old/broken UNIX systems.66 |
| **Key User Experience (Con)** | Steep learning curve for packagers.87 Debugging concretizer errors can be "scary".87 | Steep learning curve due to "anti-patterns" from "old stuff" online.37 find\_package can be complex.35 | "Overly complicated" m4 syntax.37 No viable Windows support.66 Fundamentally breaks with modern GPU compilers.11 |

### 7.1. Guidance by Programming Language

* **C++ Projects:** Use **CMake**. It is the de facto standard.36 The primary C++ developer frustration is dependency management 13, which is precisely the problem Spack solves. The **Spack + CMake** combination is the optimal solution.
* **Fortran-centric and Mixed-Language Projects:** Use **CMake**. Its native support for mixed C++/Fortran compilation is a critical feature for scientific computing.6 It automates the most difficult parts of this process (runtime linking, module order, name mangling), providing a user experience that is far superior to manual methods or Autotools.6
* **Legacy C / Shell-Script-Heavy Projects:** **Autotools** is only justifiable if the *primary* goal is portability to a wide range of obscure, legacy, or "broken" UNIX-like systems 66 and the project has *zero* GPU or Windows requirements.

### 7.2. Guidance by Project Scale and Complexity

For large-scale projects with deep, complex dependency graphs (e.g., the 70+ dependencies mentioned in 22), the **Spack + CMake** stack is the state-of-the-art solution. Spack manages the *inter-package* dependency graph via spack.yaml environments 21, while its CMakePackage integration automatically populates CMAKE\_PREFIX\_PATH.12 This allows the developer's CMakeLists.txt file to use find\_package 35 and succeed without manual intervention, cleanly separating the *developer* concern (building the application) from the *integrator* concern (building the full environment).2

### 7.3. Guidance by GPU Platform and Portability Needs

The guidance for GPU-enabled software is definitive.

* **Recommendation:** For any project targeting GPUs, **use CMake**. It is the only meta-build system with modern, "first-class" language support for CUDA 7 and HIP.8
* **Recommendation:** For any project targeting GPUs, **DO NOT use Autotools.** The user experience is demonstrably broken. It is incompatible with libtool 9 and, more fundamentally, with the NVIDIA HPC SDK's compilers.11 It is an "anti-pattern" for GPU development.
* **The Optimal Stack:** The recommended stack for E4S users is **Spack + CMake**. The developer writes a portable CMakeLists.txt 49 and the packager writes a Spack package.py inheriting CudaPackage.86 The end-user installs by specifying a simple, high-level Spack variant (e.g., +cuda, cuda\_arch=80, +rocm).24 Spack automatically translates this simple variant into the complex, correct flags for CMake, abstracting the entire build complexity.

### 7.4. Conclusions and Strategic Recommendations

Based on this analysis of the E4S 25.06 ecosystem, the following strategic recommendations are presented to guide future development and integration efforts:

1. **Standardize on CMake:** For all new scientific software development, teams should standardize on CMake as the meta-build system (Layer 2). Its robust user experience for C++, Fortran, mixed-language, and GPU-native compilation provides the best long-term viability.
2. **Deprecate Autotools for GPU Work:** For all projects targeting GPUs, mandate the use of CMake's native language support (enable\_language(CUDA/HIP)) and explicitly deprecate the use of GNU Autotools. The evidence of its systemic incompatibility with modern GPU toolchains is conclusive.
3. **Adopt Spack Universally:** Fully adopt Spack as the high-level integration and deployment tool (Layer 3). The user experience it provides for managing the combinatorial explosion of compilers, libraries, and GPU variants is the core value proposition of E4S.
4. **Focus Training on the Spack + CMake Synergy:** Invest in training that focuses on this symbiotic relationship. Developers must master "Modern CMake" 38, and packagers/admins must master Spack's package.py DSL 5 and CMakePackage integration 12 to maximize the ecosystem's benefits.
5. **Leverage the Full E4S 25.06 Lifecycle:** Promote the use of the E4S 25.06 container tools. e4s-alc 14 should be used to create minimal, application-specific containers, and e4s-cl 15 should be the standard launcher for all containerized MPI jobs on HPC facilities to ensure portability *and* native performance.

#### Works cited

1. Using Spack to Accelerate Developer Workflows, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/ecp22/_static/slides/spack-ecp22-tutorial-slides.pdf>
2. Managing HPC Software Complexity with Spack, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/latest/_static/slides/spack-radiuss23-tutorial-slides.pdf>
3. Spack 1.1.0.dev0 documentation, accessed November 6, 2025, <https://spack.readthedocs.io/>
4. Spack – scientific software package manager for supercomputers, Linux, and macOS | Hacker News, accessed November 6, 2025, <https://news.ycombinator.com/item?id=35237269>
5. Managing HPC Software Complexity with Spack, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/pearc23/_static/slides/spack-pearc23-tutorial-slides.pdf>
6. Fortran for C/C++ developers made easier with CMake - Kitware, Inc., accessed November 6, 2025, <https://www.kitware.com/fortran-for-cc-developers-made-easier-with-cmake/>
7. FindCUDA — CMake 4.2.0-rc2 Documentation, accessed November 6, 2025, <https://cmake.org/cmake/help/latest/module/FindCUDA.html>
8. enable\_language — CMake 4.2.0-rc2 Documentation, accessed November 6, 2025, <https://cmake.org/cmake/help/latest/command/enable_language.html>
9. autotools, build shared library, cuda - NVIDIA Developer Forums, accessed November 6, 2025, <https://forums.developer.nvidia.com/t/autotools-build-shared-library-cuda/24152>
10. CUDA ,C and Autotools - NVIDIA Developer Forums, accessed November 6, 2025, <https://forums.developer.nvidia.com/t/cuda-c-and-autotools/36274>
11. HPC SDK does not work with Autoconf/Automake-based projects ..., accessed November 6, 2025, <https://forums.developer.nvidia.com/t/hpc-sdk-does-not-work-with-autoconf-automake-based-projects/277894>
12. CMake - Spack 1.1.0.dev0 documentation, accessed November 6, 2025, <https://spack.readthedocs.io/en/latest/build_systems/cmakepackage.html>
13. Spack on Windows: A New Era in Cross-Platform Dependency Management - Kitware Inc., accessed November 6, 2025, <https://www.kitware.com/spack-on-windows-a-new-era-in-cross-platform-dependency-management/>
14. E4S-Project/e4s-alc: E4S à la carte is a tool that allows a user to customize a container image by adding packages to it. These can be system packages and Spack packages. - GitHub, accessed November 6, 2025, <https://github.com/E4S-Project/e4s-alc>
15. e4s-cl Demo with Trilinos Zoltan - YouTube, accessed November 6, 2025, <https://www.youtube.com/watch?v=6eZflZpHldk>
16. Spack: Package Management for HPC - ATPESC, accessed November 6, 2025, <https://extremecomputingtraining.anl.gov/wp-content/uploads/sites/96/2023/08/ATPESC-2023-Track-3-Talk-4-Becker-Spack.pdf>
17. Managing HPC Software Complexity with Spack, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/cineca23/_static/slides/spack-cineca23-tutorial-slides.pdf>
18. E4S - Home, accessed November 6, 2025, <https://e4s.io/>
19. E4S: Extreme-Scale Scientific Software Stack, accessed November 6, 2025, <https://bssw.io/items/e4s-extreme-scale-scientific-software-stack>
20. Advancing High-Performance Computing - DoD HPC Modernization Program Uses Spack and E4S, accessed November 6, 2025, <https://cass.community/impacts/2025-01-dod-hpc.html>
21. E4S and SDK Efforts - Exascale Computing Project, accessed November 6, 2025, <https://www.exascaleproject.org/research-project/e4s-and-sdk-efforts/>
22. The Spack Package Manager: Bringing Order to HPC Software Chaos - tgamblin.github.io, accessed November 6, 2025, <https://tgamblin.github.io/pubs/spack-sc15.pdf>
23. About Spack, accessed November 6, 2025, <https://spack.io/about/>
24. hipSYCL - Spack Packages, accessed November 6, 2025, <https://packages.spack.io/package.html?name=hipsycl>
25. Spack, accessed November 6, 2025, <https://spack.io/>
26. Software Deployment Process for Spack E4S stack at ... - eScholarship, accessed November 6, 2025, <https://escholarship.org/content/qt5zh5z08q/qt5zh5z08q_noSplash_7481977d801ffa476891b027d53230d4.pdf>
27. The Extreme-scale Scientific Software Stack (E4S) | GitHub website for E4S Project, accessed November 6, 2025, <https://e4s.io/manual-installation.html>
28. Providing a Flexible and Comprehensive Software Stack Via Spack, an Extreme-Scale Scientific Software Stack, and Software Development Kits - IEEE Computer Society, accessed November 6, 2025, <https://www.computer.org/csdl/magazine/cs/2024/01/10513439/1WyZfne1na0>
29. Introduction to Spack in the ADMIRE Project, accessed November 6, 2025, <https://admire-eurohpc.eu/wp-content/uploads/2023/04/E4S.pdf>
30. Build Systems - Spack 1.1.0.dev0 documentation, accessed November 6, 2025, <https://spack.readthedocs.io/en/latest/build_systems.html>
31. Spack Package Build Systems — Spack Tutorial documentation - Read the Docs, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/pearc23/tutorial_buildsystems.html>
32. Spack Package Build Systems — Spack Tutorial documentation - Read the Docs, accessed November 6, 2025, <https://spack-tutorial.readthedocs.io/en/acmrep24/tutorial_buildsystems.html>
33. Managing HPC Software Complexity with - Spack - OACISS : Oregon Advanced Computing Institute for Science and Society, accessed November 6, 2025, <https://oaciss.uoregon.edu/NSFDOE19/talks/Spack-DOE-NSF-Tutorial-2019.pdf>
34. Building With Spack | Gnosis Research Center, accessed November 6, 2025, <https://grc.iit.edu/docs/hpc-tutorials/installing-hpc-software/building-with-spack/>
35. find\_package — CMake 4.2.0-rc2 Documentation, accessed November 6, 2025, <https://cmake.org/cmake/help/latest/command/find_package.html>
36. CMake - Upgrade Your Software Build System, accessed November 6, 2025, <https://cmake.org/>
37. Autotools vs Cmake - DEV Community, accessed November 6, 2025, <https://dev.to/vimmer9/autotools-vs-cmake-44kg>
38. An Introduction to Modern CMake, accessed November 6, 2025, <https://cliutils.gitlab.io/modern-cmake/>
39. I wrote a comprehensive guide to modern CMake using a real 80-file game engine project (not another hello-world tutorial) - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/cpp/comments/1mjw02w/i_wrote_a_comprehensive_guide_to_modern_cmake/>
40. Mixing C++ and Fortran — CMake Workshop, accessed November 6, 2025, <https://enccs.github.io/cmake-workshop/cxx-fortran/>
41. Getting started with CMake for Fortran | Emily's Website 🏳️‍⚧️, accessed November 6, 2025, <https://www.atomwitch.net/2022/04/02/cmake-fortran.html>
42. Overview of ECP Software Technology - Exascale Computing Project, accessed November 6, 2025, <https://www.exascaleproject.org/wp-content/uploads/2022/05/E4S-2022-08-25-Heroux.pdf>
43. Is there any findCuda.cmake step by step tutorial? - NVIDIA Developer Forums, accessed November 6, 2025, <https://forums.developer.nvidia.com/t/is-there-any-findcuda-cmake-step-by-step-tutorial/11817>
44. CUDA and cmake - NVIDIA Developer Forums, accessed November 6, 2025, <https://forums.developer.nvidia.com/t/cuda-and-cmake/11600>
45. How to find and link CUDA libraries using CMake 3.15 and later? - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/66327073/how-to-find-and-link-cuda-libraries-using-cmake-3-15-and-later>
46. CUDA projects | CLion Documentation - JetBrains, accessed November 6, 2025, <https://www.jetbrains.com/help/clion/cuda-projects.html>
47. enable\_language — CMake 3.22.6 Documentation, accessed November 6, 2025, <https://cmake.org/cmake/help/v3.22/command/enable_language.html>
48. Using CMake - AMD ROCm documentation, accessed November 6, 2025, <https://rocm.docs.amd.com/en/latest/conceptual/cmake-packages.html>
49. Application portability with HIP - AMD GPUOpen, accessed November 6, 2025, <https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-hipify-readme/>
50. hip::host not working for NVIDIA platforms - Development - CMake ..., accessed November 6, 2025, <https://discourse.cmake.org/t/hip-host-not-working-for-nvidia-platforms/13577>
51. SYCL, CUDA, and others --- experiences and future trends in heterogeneous C++ programming? : r/cpp - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/cpp/comments/1im99l2/sycl_cuda_and_others_experiences_and_future/>
52. Portable kernel-based models — GPU programming: why, when and how? documentation, accessed November 6, 2025, <https://enccs.github.io/gpu-programming/8-portable-kernel-models/>
53. invexed/hipSYCL: Implementation of SYCL for CPUs, AMD GPUs, NVIDIA GPUs - GitHub, accessed November 6, 2025, <https://github.com/invexed/hipSYCL>
54. A collection of samples written using the SYCL standard for C++. - GitHub, accessed November 6, 2025, <https://github.com/codeplaysoftware/SYCL-samples>
55. Compiling SYCL with Different GPUs - Intel, accessed November 6, 2025, <https://www.intel.com/content/www/us/en/developer/articles/technical/compiling-sycl-with-different-gpus.html>
56. Autotools FAQ - GNU, accessed November 6, 2025, <https://www.gnu.org/software/automake/faq/autotools-faq.html>
57. GNU Autoconf, Automake and Libtool - Shlomi Fish, accessed November 6, 2025, <https://www.shlomifish.org/lecture/Autotools/slides--all-in-one-html/>
58. The basics of Autotools - Gentoo Development Guide, accessed November 6, 2025, <https://devmanual.gentoo.org/general-concepts/autotools/index.html>
59. The configure.ac File | BuildTools - GitHub Pages, accessed November 6, 2025, <https://coin-or-tools.github.io/BuildTools/configure.html>
60. Autoconf - Wikipedia, accessed November 6, 2025, <https://en.wikipedia.org/wiki/Autoconf>
61. autotools - What are Makefile.am and Makefile.in? - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/2531827/what-are-makefile-am-and-makefile-in>
62. GNU Automake, accessed November 6, 2025, <https://www.gnu.org/software/automake/manual/1.7/automake.html>
63. Difference Between Makefile.am and Makefile.in | Baeldung on Linux, accessed November 6, 2025, <https://www.baeldung.com/linux/makefile-am-vs-in>
64. Introduction to GNU Autotools | Opensource.com, accessed November 6, 2025, <https://opensource.com/article/19/7/introduction-gnu-autotools>
65. linux - Makefile.am ... what are those?, accessed November 6, 2025, <https://stackoverflow.com/questions/1637714/makefile-am-what-are-those>
66. What are the differences between Autotools, Cmake and Scons? - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/4071880/what-are-the-differences-between-autotools-cmake-and-scons>
67. Questions about cmake, make, and autotools : r/cprogramming - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/cprogramming/comments/jzb9l1/questions_about_cmake_make_and_autotools/>
68. Autotools vs CMake for both Windows and Linux compilation - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/5837764/autotools-vs-cmake-for-both-windows-and-linux-compilation>
69. Build CUDA and C++ using Autotools - Stack Overflow, accessed November 6, 2025, <https://stackoverflow.com/questions/23116674/build-cuda-and-c-using-autotools>
70. E4S: The Extreme-scale Scientific Software Stack Release 24.11, accessed November 6, 2025, <https://oaciss.uoregon.edu/e4s/talks/E4S_24.11.pdf>
71. E4S: The Extreme-scale Scientific Software Stack Release 24.05, accessed November 6, 2025, <https://oaciss.uoregon.edu/e4s/talks/E4S_24.05.pdf>
72. E4S - NERSC Documentation, accessed November 6, 2025, <https://docs.nersc.gov/applications/e4s/>
73. E4S — Much More than Just the Delivery Vehicle for Hardened and ..., accessed November 6, 2025, <https://www.exascaleproject.org/highlight/e4s-much-more-than-just-the-delivery-vehicle-for-hardened-and-robust-hpc-libraries-and-tools/>
74. accessed November 6, 2025, <https://github.com/E4S-Project/e4s-alc#:~:text=Overview,to%20generate%20images'%20definition%20files.>
75. accessed November 6, 2025, <https://e4s.io/e4s-cl.html#:~:text=E4S's%20container%20management%20tool%3A%20E4S,new%20binary%20on%20the%20host.>
76. E4S-CL, accessed November 6, 2025, <https://e4s.io/e4s-cl.html>
77. E4S Container Launcher 1.0.5 documentation, accessed November 6, 2025, <https://e4s-cl.readthedocs.io/>
78. What is E4S? - Exascale Computing Project, accessed November 6, 2025, <https://www.exascaleproject.org/wp-content/uploads/2022/05/E4S-2022-08-25-Shende_1.pdf>
79. E4S Overview and Demo - OACISS : Oregon Advanced Computing Institute for Science and Society, accessed November 6, 2025, <https://oaciss.uoregon.edu/E4S-Forum20/talks/Shende_E4S_Demo_Sep20.pdf>
80. E4S: Extreme-scale Scientific Software Stack - Collegeville, accessed November 6, 2025, <https://collegeville.github.io/CW20/WorkshopResources/WhitePapers/heroux-willenbring-shende-coti-spear-et-al-E4S.pdf>
81. CMake 4.2.0-rc2 Documentation, accessed November 6, 2025, <https://cmake.org/cmake/help/latest/index.html>
82. Introduction to CMake — Introduction to CMake documentation, accessed November 6, 2025, <https://enccs.github.io/intro-cmake/>
83. Programming tips, C++, Fortran CMake. Programming my own solver : r/CFD - Reddit, accessed November 6, 2025, <https://www.reddit.com/r/CFD/comments/18l64vn/programming_tips_c_fortran_cmake_programming_my/>
84. Autotools Tutorial - Drexel University, accessed November 6, 2025, <https://faculty.coe.drexel.edu/jwalsh/Mengkeautotools.pdf>
85. GNU Autotools: a tutorial - Bootlin, accessed November 6, 2025, <https://bootlin.com/pub/conferences/2016/elc/petazzoni-autotools-tutorial/petazzoni-autotools-tutorial.pdf>
86. Cuda - Spack 1.1.0.dev0 documentation, accessed November 6, 2025, <https://spack.readthedocs.io/en/latest/build_systems/cudapackage.html>
87. Losing Users: Perception that Spack is Too Complicated · Issue #2873 - GitHub, accessed November 6, 2025, <https://github.com/spack/spack/issues/2873>
88. Better Out-of-the-box CMake Support · Issue #7904 · spack/spack - GitHub, accessed November 6, 2025, <https://github.com/spack/spack/issues/7904>