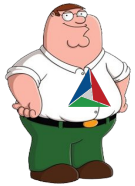
Modern CMake

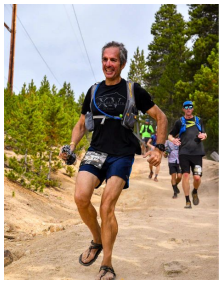
Past, Present, and Future

BILL HOFFMAN

About Me

- **1990-1999 GE research in a Computer Vision group.**
  - Moved them from Symbolics Lisp Machines to C++ (CFront) on Solaris and later Linux/Windows
  - Was the build and software library guy (gmake/autotools)
- **1998-Present Founder and CTO Kitware**
  - Started CMake in 1999
  - Mostly management now (pointy haired boss stuff), finding funding for CMake





kitware

What is a Kitware and what does it do?

kitware

What is a Kitware and what does it do?

- Collects $100 from everyone that uses CMake from the CMake subscription model

Let's ask chatgpt as we do these days

Let’s estimate step by step.

1. **Figure out how many CMake users exist.**
   - CMake is one of the most widely used build systems in C/C++.
   - It’s open source, so there’s no exact count of users.
   - Indicators:
     - ~60k GitHub stars.
     - Bundled with major Linux distributions and used by countless developers.
     - Likely at least **a few million developers** worldwide use it directly or indirectly.
2. **Rough scenarios:**

Estimated Users	$100 each/year	Total per year
1 million	$100 × 1,000,000 = $100,000,000	$100 million
2 million	$100 × 2,000,000 = $200,000,000	$200 million
5 million	$100 × 5,000,000 = $500,000,000	$500 million
10 million	$100 × 10,000,000 = $1,000,000,000	$1 billion

3. **Conclusion:**

   Depending on whether CMake has 1 million or 10 million+ active users, you’d collect somewhere between **$100 million and $1 billion per year** at $100 per user annually.

How does Kitware spend between $100 million and $1 billion of CMake revenue?

With **$100 million/year**, you could realistically hire:

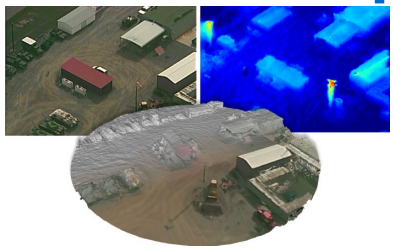
- **~500 senior developers** in the U.S./Western Europe, or
- **~1,000 developers** in mixed regions at moderate cost, or
- **2,000+ developers** in lower-cost regions.

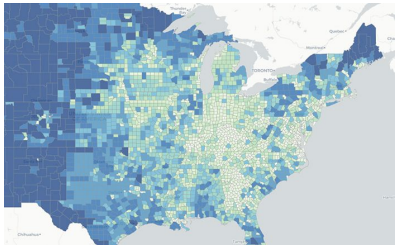
kitware

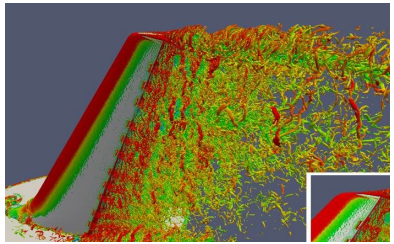
What do the 1000 CMake developers at Kitware do

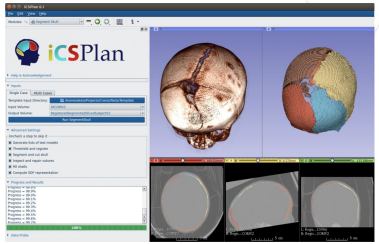
- Develop any feature that the users want
- Fix any issue reported in 24 hours
- Write perfect documentation
- Provide support to any question

Kitware actually does consulting and research contracts: Areas of expertise / Built on open source









Computer Vision

Data and Analytics

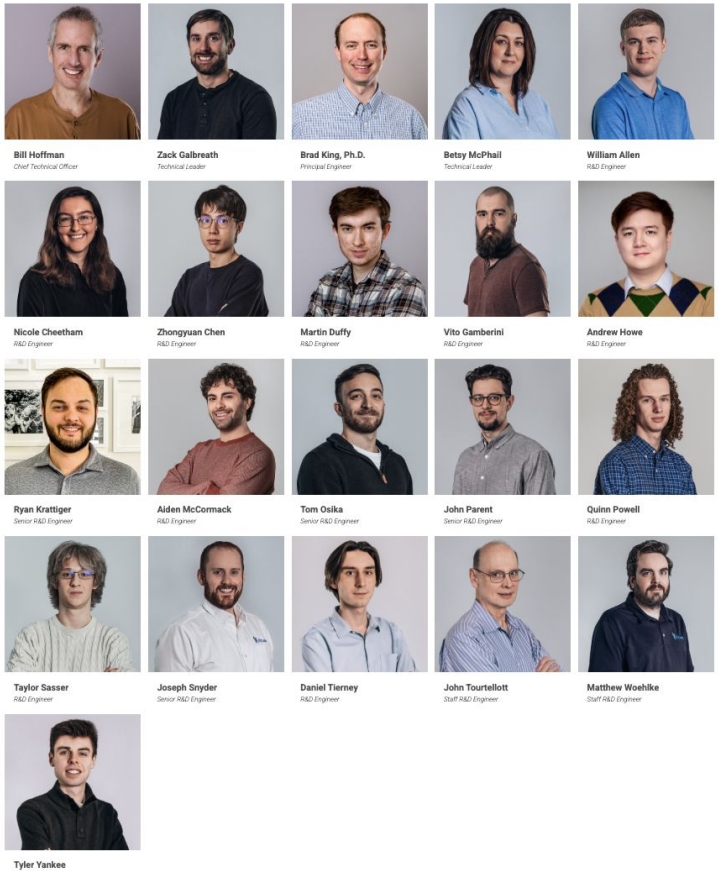
Scientific Computing

Medical Computing

Software Solutions

kitware

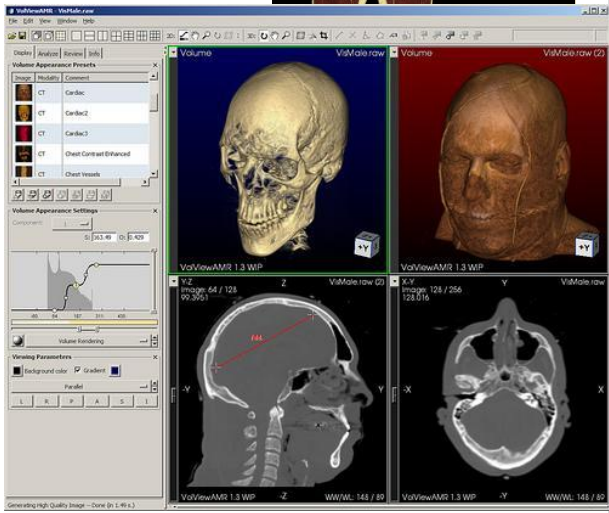
Kitware Software Solutions Team (CMakeish team)



Small team in 200+ person company

Where did CMake come from?

- Kitware was the lead engineering team for the Insight Segmentation and Registration Toolkit (ITK) http://www.itk.org
  - tasked with making it build on Unix/Windows/Mac
- Funded by National Library of Medicine (NLM): part of the Visible Human Project
  - Data CT/MR/Slice 1994/1995
  - Code (ITK) 1999
- CMake Release-1-0 branch created in 2001
- Since then funding from many projects and outside contributions



kitware

Happy Birthday CMake!

August 31, 2015 Bill Hoffman

The CMake project officially launched on this day in 2000 on the Insight Segmentation and Registration Toolkit (ITK) mailing list:

http://public.kitware.com/pipermail/insight-developers/2000-August/024248.html

A small, five-person company, called Kitware (here is our 2000 web page), was taking part in a project funded by the National Library of Medicine to create a cross-platform C++ toolkit that would codify the state-of-the-art in segmentation and registration algorithms and complement the data in the Visible Human project. One of Kitware's tasks was to create a build system that would work on Unix, Windows, and Mac OS. Kitware's founding project, VTK, had a build system called pcmaker, which used autotools on Unix and a C++ program that created nmake makefiles on Windows. CMake was created with the limitations of that tool in mind.

One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to use.

25 Years! Yikes, I was 33!

Happy Birthday CMake!

August 31, 2015 Bill Hoffman

The CMake project officially launched on this day in 2000 on the Insight Segmentation and Registration Toolkit (ITK) mailing list:

http://public.kitware.com/pipermail/insight-developers/2000-August/024248.html

A small, five-person company, called Kitware (here is our 2000 web page), was taking part in a project funded by the National Library of Medicine to create a cross-platform C++ toolkit that would codify the state-of-the-art in segmentation and registration algorithms and complement the data in the Visible Human project. One of Kitware's tasks was to create a build system that would work on Unix, Windows, and Mac OS. Kitware's founding project, VTK, had a build system called pcmaker, which used autotools on Unix and a C++ program that created nmake makefiles on Windows. CMake was created with the limitations of that tool in mind.

One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to use.

kitware

about solutions expertise news careers contact EU Enter Keyword

news › blog › post

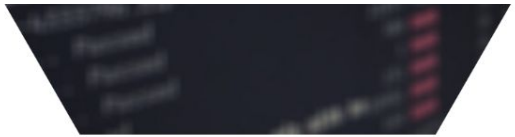
25 Years! Yikes, I was 33!

Happy Birthday CMake!

August 31, 2015 Bill Hoffman

One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to use.

registration algorithms and complement the data in the Visible Human project. One of Kitware's tasks was to create a build system that would work on Unix, Windows, and Mac OS. Kitware's founding project, VTK, had a build system called pcmaker, which used autotools on Unix and a C++ program that created nmake makefiles on Windows. CMake was created with the limitations of that tool in mind.



One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to

Don't forget to SUBSCRIBE.

Happy Birthday CMake!

August 31, 2015 Bill Hoffman

The CMake project officially launched on this day in 2000 on the Insight Segmentation and Registration Toolkit (ITK) mailing list:

http://public.kitware.com/pipermail/insight-developers/2000-August/024248.html

A small, five-person company, called Kitware (here is our 2000 web page), was taking part in a project funded by the National Library of Medicine to create a cross-platform C++ toolkit that would codify the state-of-the-art in segmentation and registration algorithms and complement the data in the Visible Human project. One of Kitware's tasks was to create a build system that would work on Unix, Windows, and Mac OS. Kitware's founding project, VTK, had a build system called pcmaker, which used autotools on Unix and a C++ program that created nmake makefiles on Windows. CMake was created with the limitations of that tool in mind.

One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to use.

kitware

news › blog › post

Happy Birthday C

August 31, 2015 Bill Hoffm

The CMake project officially launched
list:

http://public.kitware.com/pipermail/in

A small, five-person company, called K
Library of Medicine to create a cross-p
registration algorithms and compleme
system that would work on Unix, Winc
which used autotools on Unix and a C
limitations of that tool in mind.

One of the main objectives of CMak
+, the only thing on which we could
that it can be built on any system on v
hunt build process. First find/build/in
use.

kitware

public.kitware.com/pipermail/insight-developers/2000-August/024248.html

Kitware Bookmarks | Kitware - Calendar... | Login | 1Password... | Inbox (22) - bill.ho... | K

[Insight-developers] CMake / ITK build process

Bill Hoffman bill.hoffman at kitware.com
Thu Aug 31 18:29: [Insight-developers] CMake / ITK build process

- Previous message: [Insight-developers] itkBasicArchitectureTest fails on execut ion
- **Messages sorted by:** [date] [thread] [subject] [author]

I have reworked the build process for ITK.
I have created a package called CMake (Cross platform make).
It is in Insight/CMake, and it includes all the makefile.in
and c++ source for the build process of ITK.

I have also updated and integrated the install documentation with the
InsightStart.doc in Insight/Documents. The InsightStart.doc
is also now available on the main ITK web page from the
top level (as a link):

http://public.kitware.com/Insight/HTMLPrivate/

Here is the direct link to InsightStart on the web:
http://public.kitware.com/Insight/HTMLPrivate/InsightStart.html

(BTW, You can use your cvs login and password to get into the private
web page area.)

The build process is almost the same as it was. However
a few names have changed (pcbuilder -> CMakeSetup).
Also, Makefile.in is no longer used to list the files you want
built. The file used is CMakeLists.txt, which can no be found
in place of the old Makefile.in files.

As always, let me know if there are any problems.
I still have not fixed the Release build on windows.

-Bill

EU Enter Keyword

n't forget to
BSCRIBE.

Eras of CMake

- CMake 1: Primordial Flag Soup (2000)
- CMake 2: World Domination (2004)
- CMake 3: I hope you like target_* commands (2014)
- CMake 4: ??? (2025)

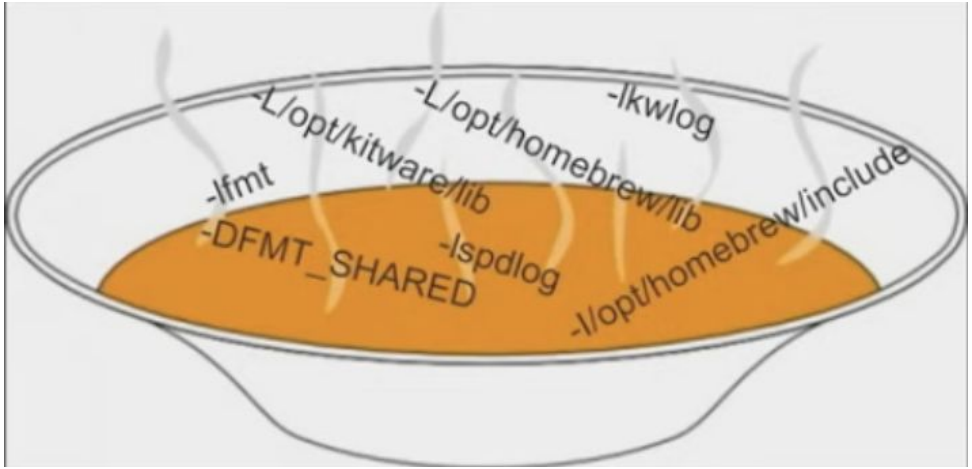
Post-Modern Cmake - From 3.0 to 4.0 - Vito Gamberini - C++Now 2025

CppNow
35.9K subscribers
Subscribe
203
Share
Save
Ask

kitware

CMake 1: Primordial Flag Soup (2000-2004)

- Friends and family era
- ITK, VTK, ParaView, a few more
- Get a bunch of compiler flags and pass them down the source tree



kitware

CMake 1: Primordial Flag Soup

```
1  SOURCE_FILES(SRCS
2  cmake
3  #...
4  cmSourceGroup
5  cmakemain
6  )
7
8  IF (WIN32)
9    SOURCE_FILES(SRCS cmDSWWriter cmDSPWriter cmMSProjectGenerator)
10 ELSE (WIN32)
11   SOURCE_FILES(SRCS cmUnixMakefileGenerator)
12 ENDIF (WIN32)
13
14 ADD_EXECUTABLE(cmake SRCS)
15
16 ADD_TEST(burn cmake)
17
18 INSTALL_TARGETS(/bin cmake)
```

CMake 1: Primordial Flag Soup

```
1  SOURCE_FILES(SRCS
2  cmake
3  #...
4  cmSourceGroup
5  cmakemain
6  )
7
8  IF (WIN32)
9    SOURCE_FILES(SRCS cmDSWWriter cmDSPWriter cmMSProjectGenerator)
10 ELSE (WIN32)
11   SOURCE_FILES(SRCS cmUnixMakefileGenerator)
12 ENDIF (WIN32)
13
14 ADD_EXECUTABLE(cmake SRCS)
15
16 ADD_TEST(burn cmake)
17
18 INSTALL_TARGETS(/bin cmake)
```

Deprecated 25 years later
in CMake 4.0

But also…

- `find_file()`
- `find_library()`
- `find_package()`
- `find_path()`
- `find_program()`

kitware

CMake 2: World Domination (2004-2014)

- Me fishing for projects to use CMake

CMake 2: World Domination (2004-2014)

- Bill fishing
  use CMak

kitware

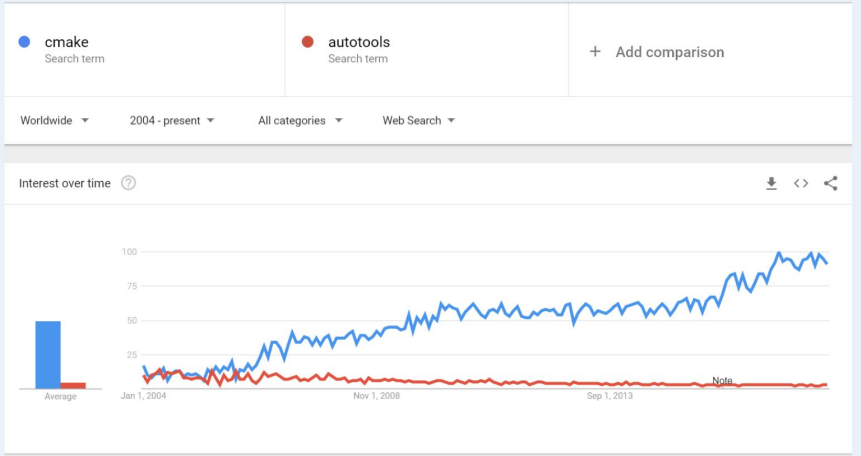
CMake 2: World Domination (2004-2014)

- 2006 KDE
  - install improvements (shared library version links, etc)
  - (cmake) pre-install on linux
  - 2012 Modern CMake
- 2008 LLVM CMake

2006

KDE switches to CMake

- KDE developer Alexander Neundorf was the champion that helped KDE adopt CMake.
- Kitware worked hard to create a prototype build system in a few short weeks.
- CMake was able to quickly build more of KDE than the scons system they were using previously.
- KDE was able to port applications to Windows and Apple very quickly with CMake.
- CMake got new features, including shared library versioning and installation rpath re-writes without relinking.
- CMake was distributed by all major linux distributions to support KDE.



kitware

CMake 2: World Domination (2004-2014)

2012 Modern CMake by Stephen Kelly

found here. Then already in 2013, the simple example with Qt shows the essence of Modern CMake:

```
find_package(Qt5Widgets 5.2 REQUIRED)

add_executable(myapp main.cpp)
target_link_libraries(myapp Qt5::Widgets)
```

steveire.wordpress.com/2017/11/05/embracing-modern-cmake/

Kitware - Calendar... | Login | 1Password... | Inbox (22) - bill.ho... | KWiK3 (beta) | Dashboard | Strava | BillHoffmanTrainin... | 0

Design a site like this with WordPress.com Get started

Dreamer, but not the only one.

« Boost dependencies and bcp API Changes in Clang »

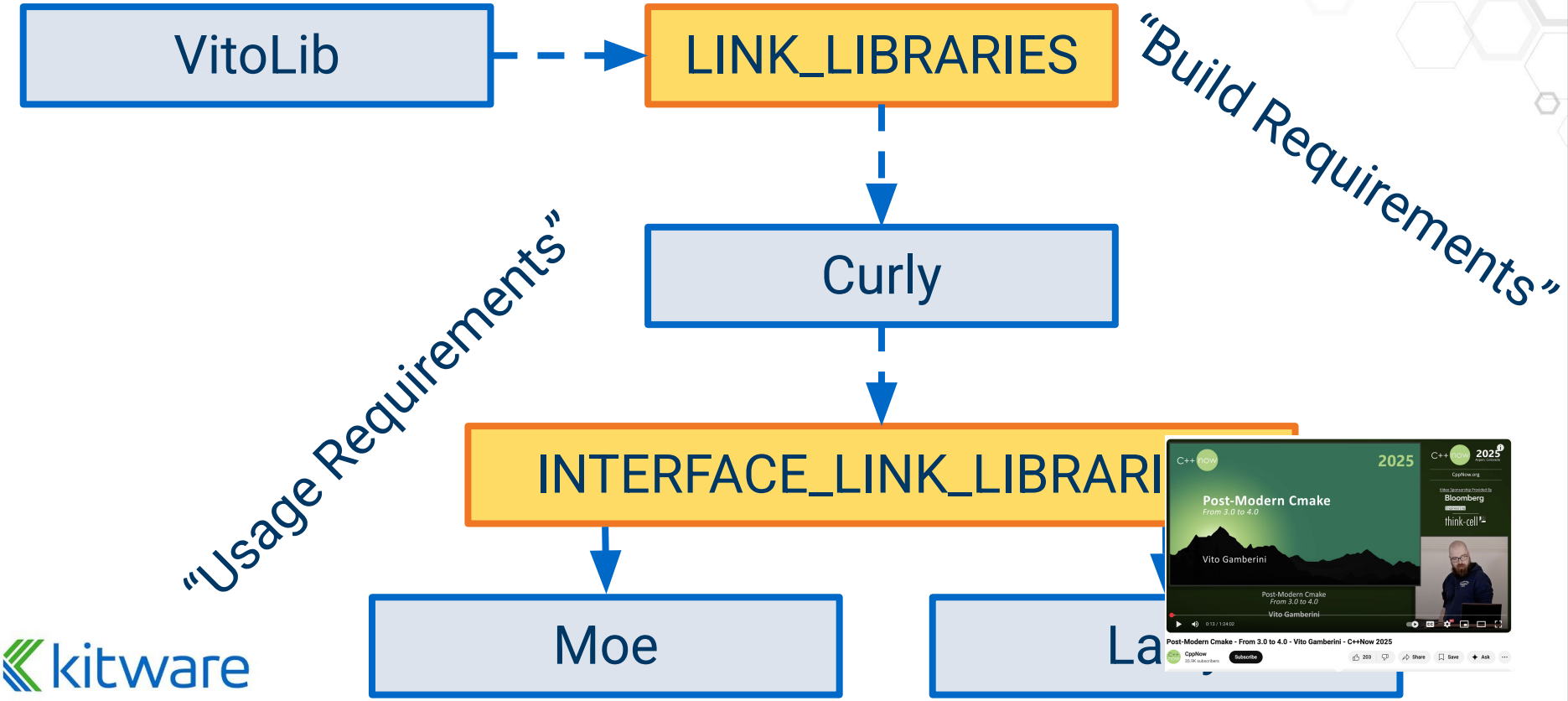
Embracing Modern CMake

I spoke at the ACCU conference in April 2017 on the topic of Embracing Modern CMake. The talk was very well attended and received, but was unfortunately not recorded at the event. In September I gave the talk again at the Dublin C++ User Group, so that it could be recorded for the internet.

The slides are available here. The intention of the talk was to present a 'gathered opinion' about what **Modern CMake** is and how it should be written. I got a lot of input from CMake users on reddit which informed some of the content of the talk.

kitware

CMake 3: I hope you like `target_*` commands (2014-2025)



kitware

CMake 3 random things I think are cool

kitware

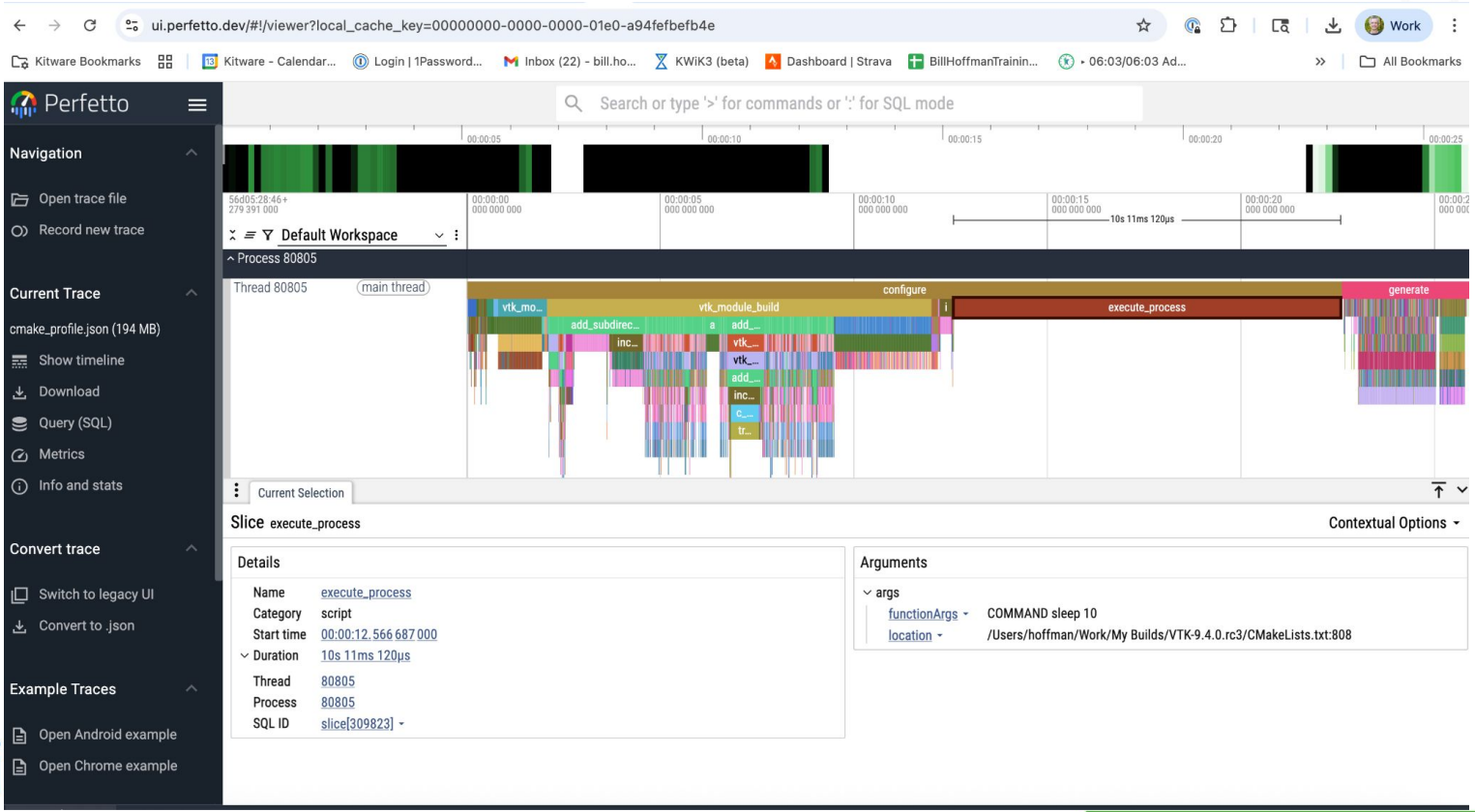
Performance testing CMake 3

```
# add some SLOW to VTK's CMakeLists.txt
#-----------------------------------------------------------------------------
# Do this at the end so the all variables it uses are setup.
include(vtkBuildPath)
execute_process(COMMAND sleep 10)
```

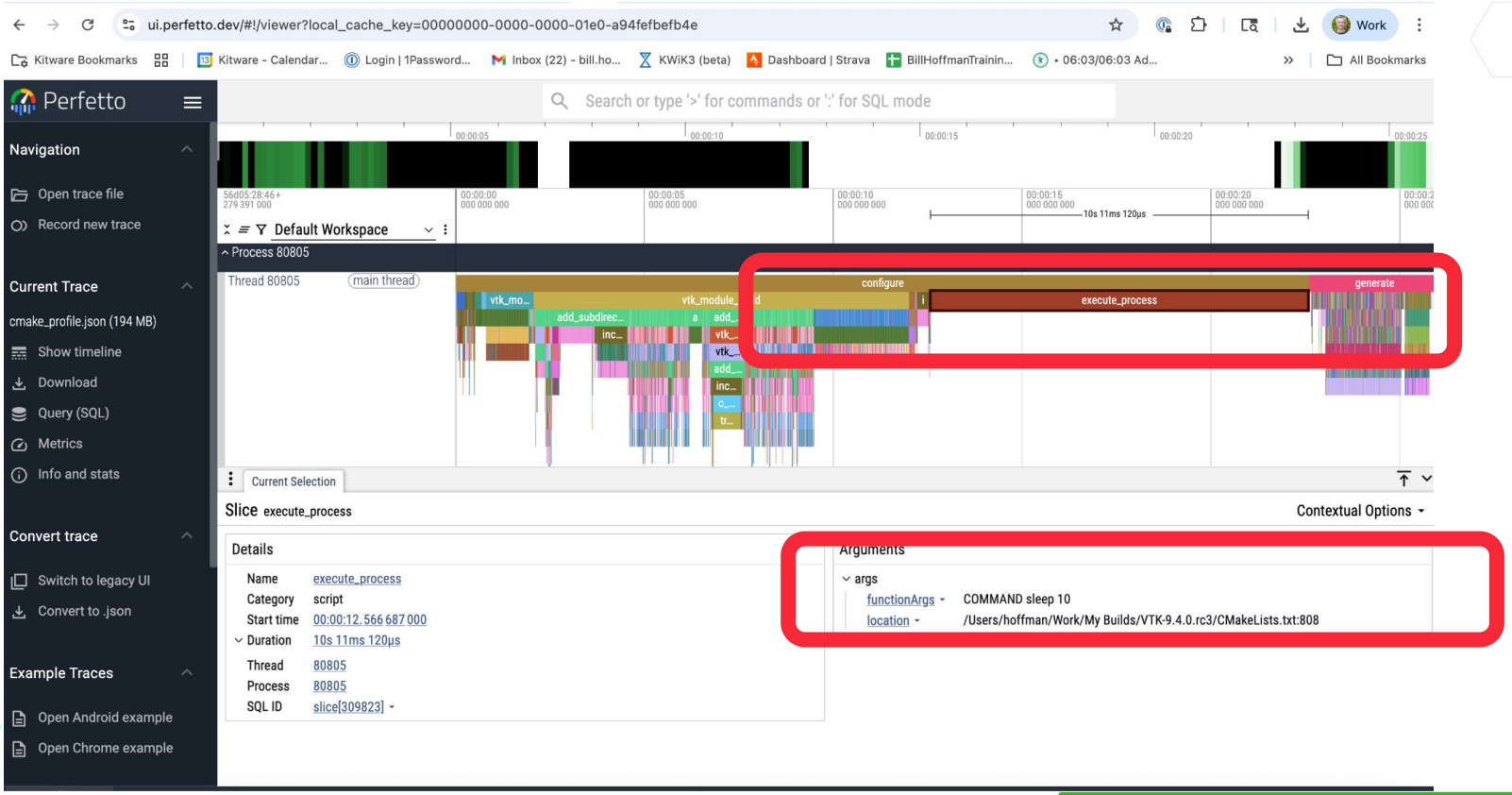
```
# run cmake and ask it for profile information
cmake  --profiling-format=google-trace --profiling-output=cmake_profile.json
```

kitware

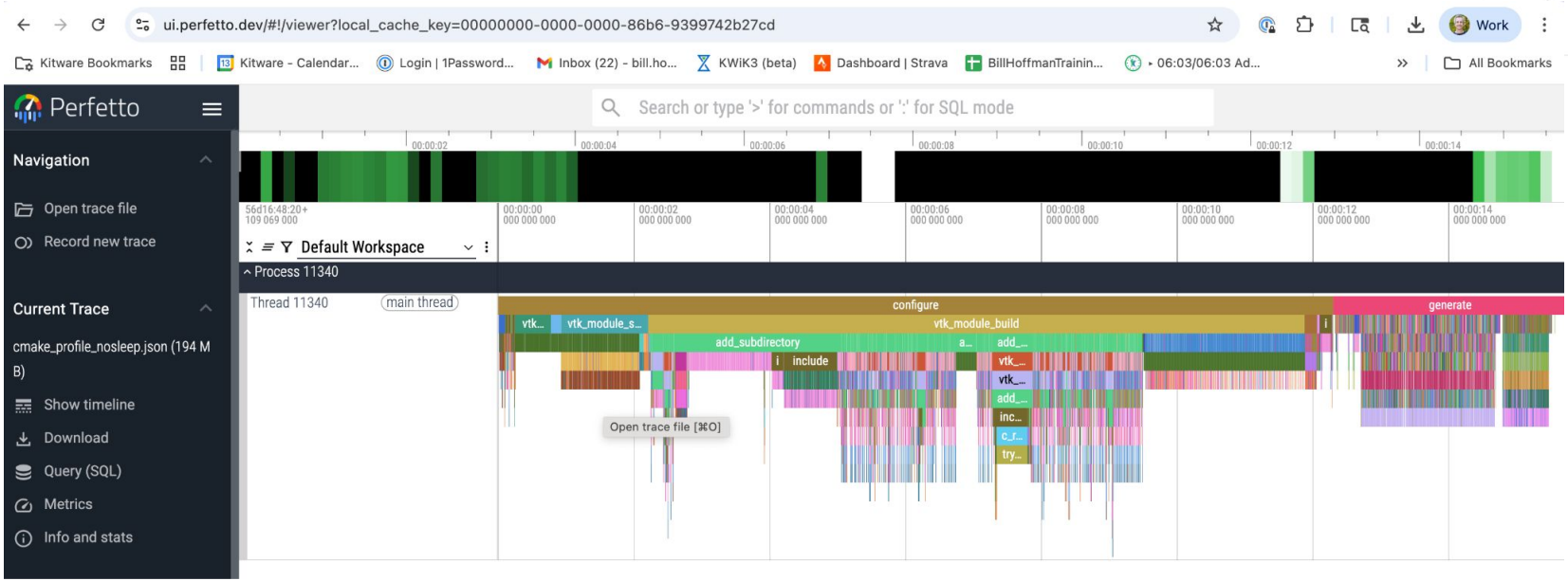
Visualize the data



Visualize the data



Remove sleep and retry

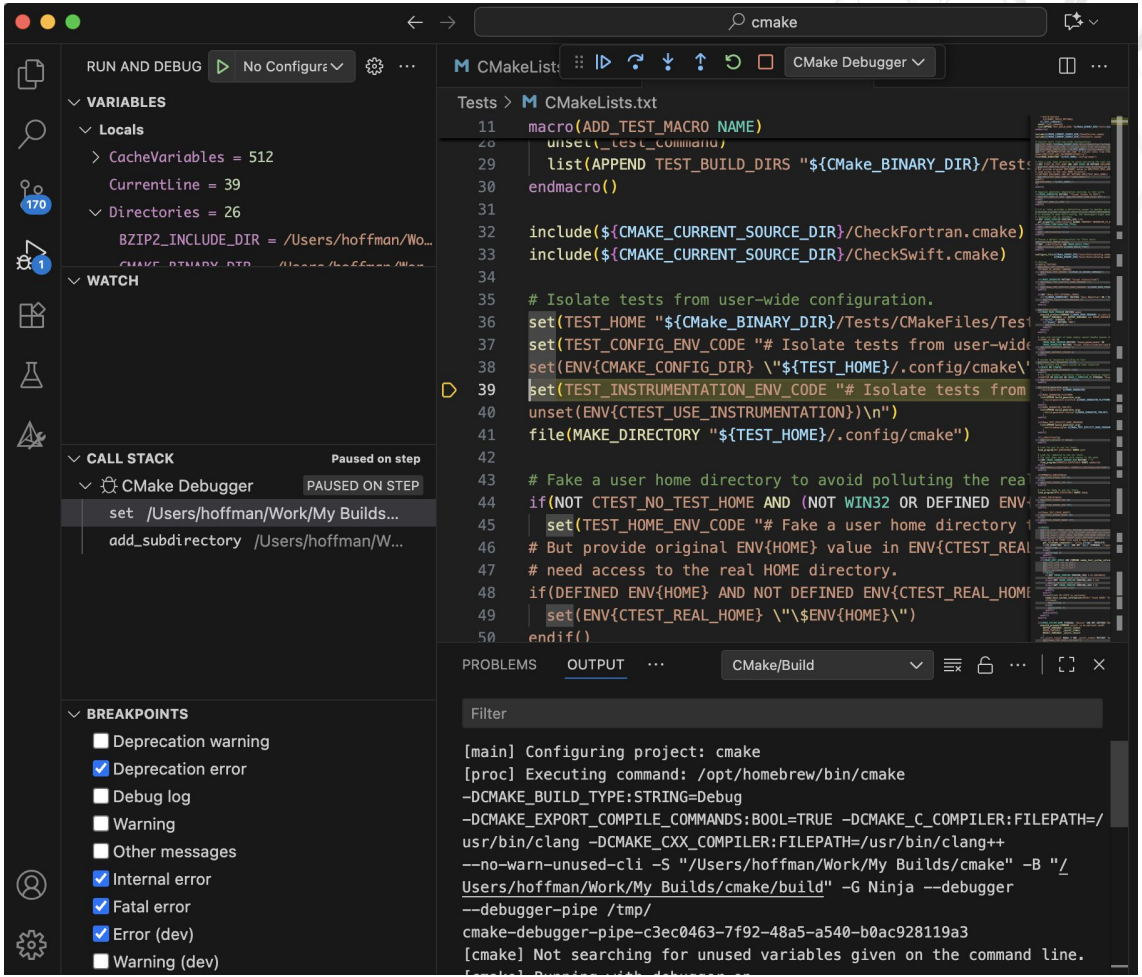


kitware

Debugging CMake 3 via Debug Adaptor Protocol (DAB)

Any editor that supports DAB can debug CMake code.

(VSCode shown)



kitware

Parallel Install (3.31 feature)

- Project developers should enable the parallel install option if the install scripts for each subdirectory can run independently in parallel

```
set_property(GLOBAL PROPERTY INSTALL_PARALLEL ON)
```

```
$ cmake --install . -j 16
```

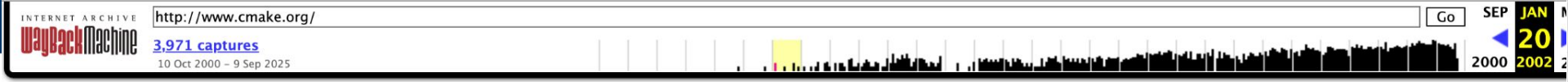
```
$ ninja install/parallel -j 16
```

Visualization of installing CMake itself in parallel using data collected by CMake Instrumentation

kitware

Before moving on to CMake 4

kitware



CMake

Cross-platform Make

- Home
- About
- Features
- Sponsors
- Download
- Install
- Example
- Documentation
- FAQ
- Mailing Lists
- Testing
- Testing Setup
- News

Welcome to CMake, the cross-platform, open-source make system. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated: it is possible to support complex environments requiring system configuration, pre-processor generation, code generation, and template instantiation. Please go here to learn more about CMake.

CMake was developed by Kitware as part of the NLM Insight Segmentation and Registration Toolkit project. The ASCI VIEWS project also provided support in the context of their parallel computation environment. Other sponsors include the Insight, VTK, and VXL open source software communities.

The goals for CMake include the following:

- Develop an open source, cross-platform tool to manage the build process,
- Allow the use of native compilers and systems,
- Simplify the build process,
- Optionally provide a user-interface to manage the build system,
- Create an extensible framework,
- Grow a self-sustaining community of software users and developers.

Earliest CMake webpage 2002

CMake

Cross-platform Make

- Home
- About
- Features
- Sponsors
- Download
- Install
- Example
- Documentation
- FAQ
- Mailing Lists
- Testing
- Testing Setup
- News

Welcome to CMake, the cross-platform, open-source make system. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated: it is possible to support complex environments requiring system configuration, pre-processor generation, code generation, and template instantiation. Please go here to learn more about CMake.

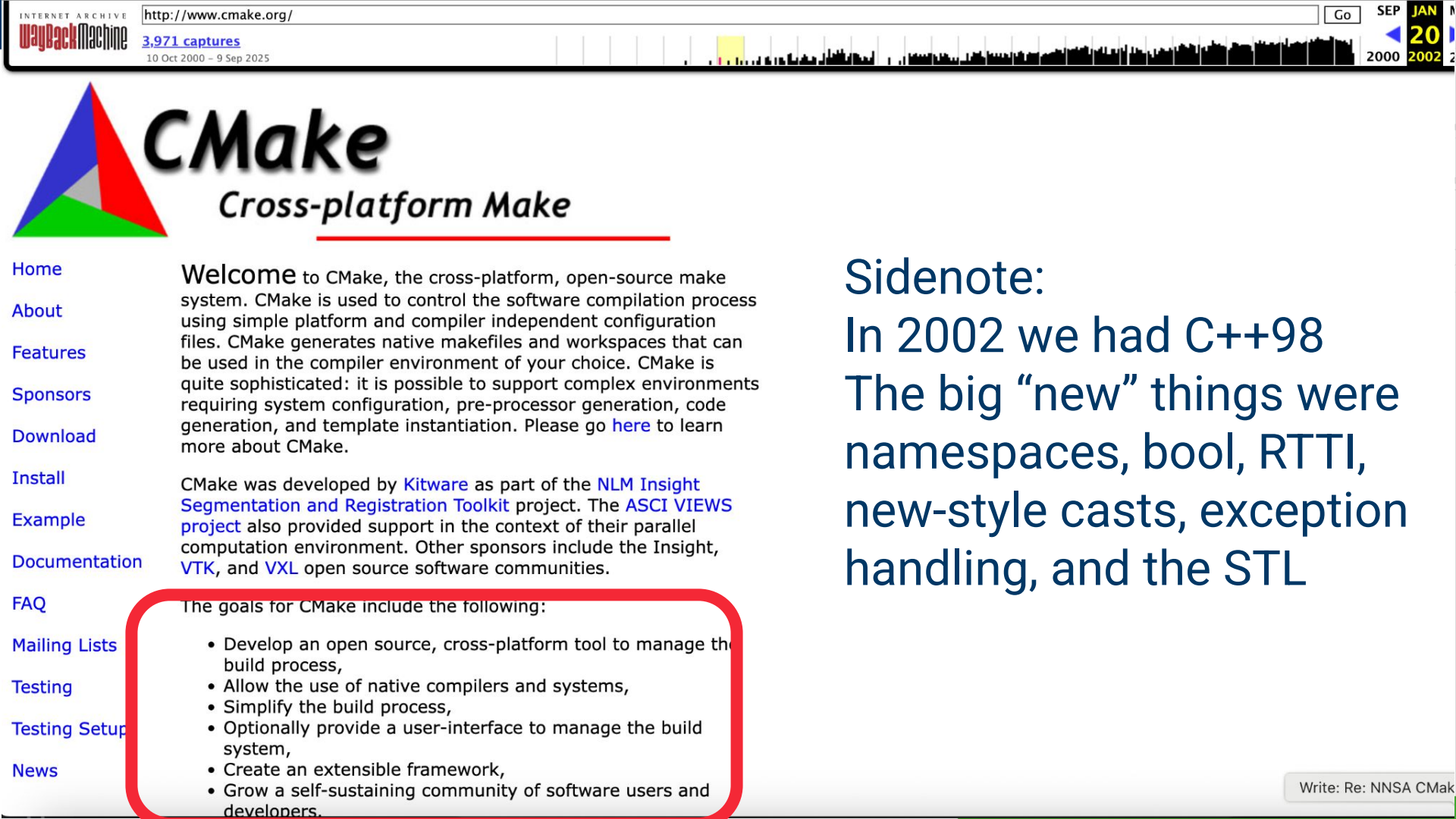
CMake was developed by Kitware as part of the NLM Insight Segmentation and Registration Toolkit project. The ASCI VIEWS project also provided support in the context of their parallel computation environment. Other sponsors include the Insight, VTK, and VXL open source software communities.

The goals for CMake include the following:

- Develop an open source, cross-platform tool to manage th build process,
- Allow the use of native compilers and systems,
- Simplify the build process,
- Optionally provide a user-interface to manage the build system,
- Create an extensible framework,
- Grow a self-sustaining community of software users and developers.

Earliest CMake webpage 2002

Write: Re: NNSA CMak



Sidenote:
In 2002 we had C++98
The big “new” things were namespaces, bool, RTTI, new-style casts, exception handling, and the STL

CMake

Cross-platform Make

Home
About
Features
Sponsors
Download
Install
Example
Documentation
FAQ
Mailing Lists
Testing
Testing Setup
News

Welcome to CMake, the cross-platform, open-source make system. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice. CMake is quite sophisticated: it is possible to support complex environments requiring system configuration, pre-processor generation, code generation, and template instantiation. Please go here to learn more about CMake.

CMake was developed by Kitware as part of the NLM Insight Segmentation and Registration Toolkit project. The ASCI VIEWS project also provided support in the context of their parallel computation environment. Other sponsors include the Insight, VTK, and VXL open source software communities.

The goals for CMake include the following:

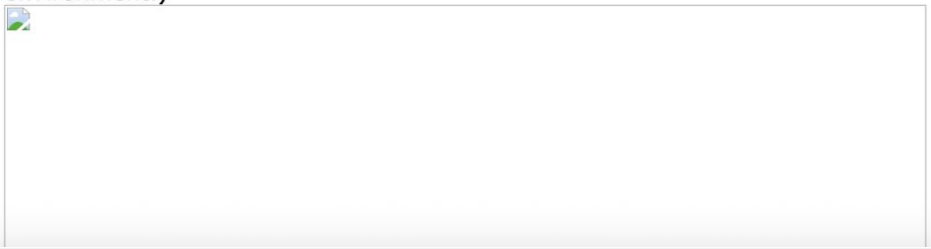
- Develop an open source, cross-platform tool to manage the build process,
- Allow the use of native compilers and systems,
- Simplify the build process,
- Optionally provide a user-interface to manage the build system,
- Create an extensible framework,
- Grow a self-sustaining community of software users and developers.

CMake

Cross-platform Make

Home
About
Features
Sponsors
Download
Install
Example
Documentation
FAQ
Mailing Lists
Testing
Testing Setup
News

CMake is an extensible, open-source system that manages the build process in an operating system and compiler independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can compile source code, create libraries, generate wrappers, and build executables in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree. CMake also supports static and dynamic library builds. Another nice feature of CMake is that it generates a cache file that is designed to be used with a graphical editor. For example, when CMake runs, it locates include files, libraries, and executable, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files. (The following figure is the CMake cache GUI in the Windows MSVC environment.)

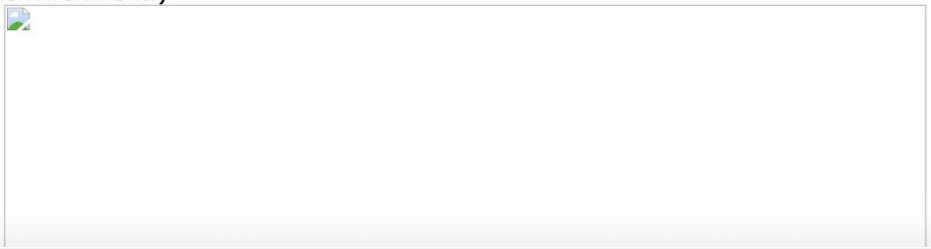


CMake

Cross-platform Make

Home
About
Features
Sponsors
Download
Install
Example
Documentation
FAQ
Mailing Lists
Testing
Testing Setup
News

CMake is an extensible, open-source system that manages the build process in an operating system and compiler independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can compile source code, create libraries, generate wrappers, and build executables in arbitrary combinations. CMake supports in-place and out-of-place builds, and can therefore support multiple builds from a single source tree. CMake also supports static and dynamic library builds. Another nice feature of CMake is that it generates a cache file that is designed to be used with a graphical editor. For example, when CMake runs, it locates include files, libraries, and executable, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files. (The following figure is the CMake cache GUI in the Windows MSVC environment.)



CMake is designed to support complex directory hierarchies and applications dependent on several libraries. For example, CMake supports projects consisting of multiple toolkits (i.e., libraries), where each toolkit might contain several directories, and the application depends on the toolkits plus additional code. CMake can also handle situations where executables must be built in order to generate code that is then compiled and linked into a final application. Because CMake is open source, and has a simple, extensible design, CMake can be extended as necessary to support new features.

Using CMake is simple. The build process is controlled by creating one or more CMakeLists.txt files in each directory (including subdirectories) that make up a project. Each CMakeLists.txt consists of one or more commands. Each command has the form COMMAND (args...) where COMMAND is the name of the command, and args is a white-space separated list of arguments. CMake provides many pre-defined commands, but if you need to, you can add your own commands. In addition, the advanced user can add other makefile generators for a particular compiler/OS combination. (While Unix and MSVC++ is supported currently, other developers are adding other compiler/OS support.) You may wish to study the examples page to see more details.

The Origins of CMake

CMake was created in response to the need for a powerful, cross-platform build environment for the Insight Segmentation and Registration Toolkit (ITK) funded by NLM as part of the Visible Human Project. It was influenced by an earlier system called **pcmaker** created by Ken Martin and other developers to support the Visualization Toolkit (VTK) open source 3D graphics and visualization system. To create CMake, Bill Hoffman at Kitware incorporated some key ideas from pcmaker, and added many more of his own, with the thought to adopt some of the functionality of the Unix **configure** tool. The initial CMake implementation was mid-2000, with acclerated development occuring in early 2001. Many improvements were due to the influences of other developers incorporating CMake into their own systems. For example, the VXL software community adopted CMake as their build environment, contributing many essential features. Brad King added several features in order to support

generators for a particular compiler/OS combination. (While Unix and MSVC++ is supported currently, other developers are adding other compiler/OS support.) You may wish to study the examples page to see more details.

The Origins of CMake

CMake was created in response to the need for a powerful, cross-platform build environment for the Insight Segmentation and Registration Toolkit (ITK) funded by NLM as part of the Visible Human Project. It was influenced by an earlier system called **pcmaker** created by Ken Martin and other developers to support the Visualization Toolkit (VTK) open source 3D graphics and visualization system. To create CMake, Bill Hoffman at Kitware incorporated some key ideas from pcmaker, and added many more of his own, with the thought to adopt some of the functionality of the Unix **configure** tool. The initial CMake implementation was mid-2000, with acclerated development occuring in early 2001. Many improvements were due to the influences of other developers incorporating CMake into their own systems. For example, the VXL software community adopted CMake as their build environment, contributing many essential features. Brad King added several features in order to support the CABLE, and automated wrapping environment and GCC-XML, and GE Corporate R&D; required support of their testing infrastructure (DART). Other features were added to support the transition of VTK's build environment to CMake, and to support ParaView, a parallel visualization system to support the Advanced Computing Lab at Los Alamos National Laboratory.

How Do I Learn CMake?

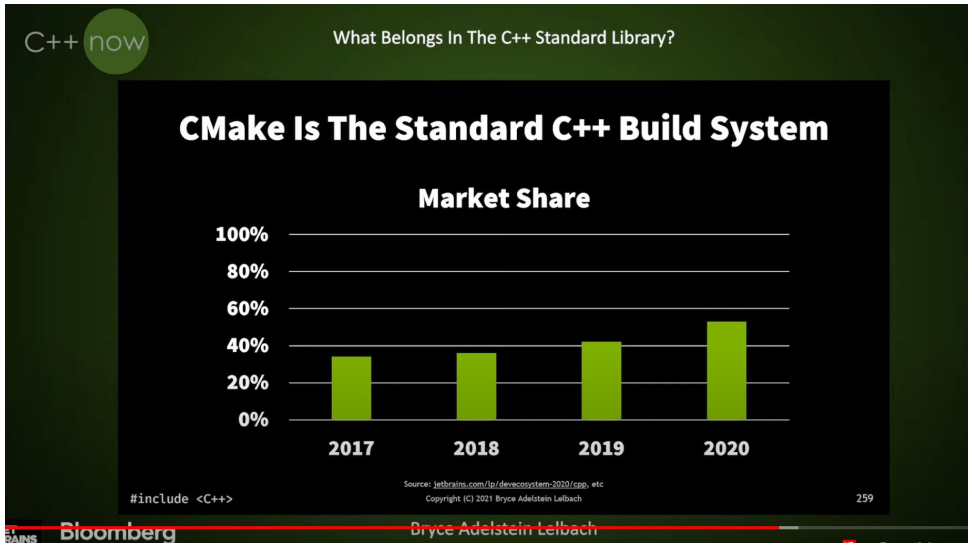
tensive use in very large software systems such as VTK, VXL, and ITK, and is therefore, quite stable. Unfortuneately, the pace of development has outstripped the pace of documentation. As a result, the best way to learn about CMake is to study existing CMake installations, and to rely on the CMake mailing list. There is some existing documentation, but until later in 2001 the documentation should be treated with caution. Please go to the documentation ple found here.

Review season, reflecting on goals

- Develop an open source, cross-platform tool to manage the build process
- Allow the use of native compilers and systems
- Simplify the build process
- Optionally provide a user-interface to manage the build system
- Create an extensible framework
- Grow a self-sustaining community of software users and developers.

Develop an open source, cross-platform tool to manage the build process

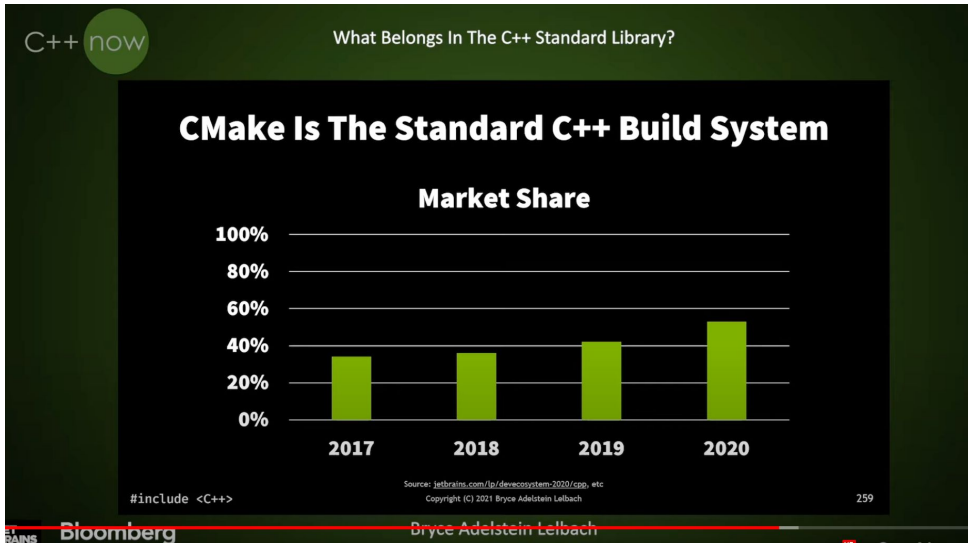
"you want a standard C++ build system, you got one it's called CMake, resistance is futile just use CMake" -
Bryce Adelstein C++ Now Jul 2021



kitware

Develop an open source, cross-platform tool to manage the build process

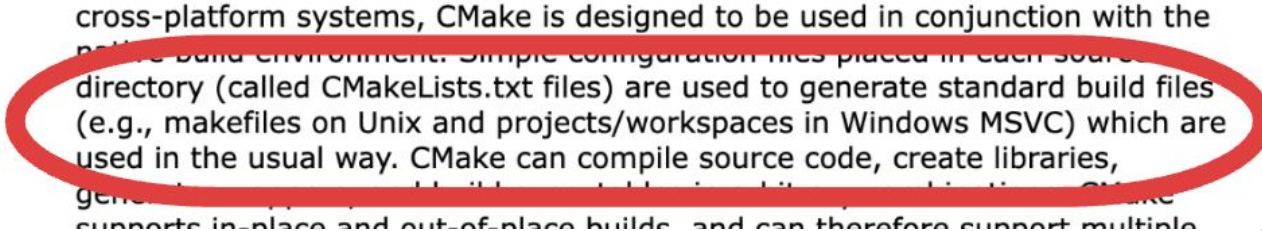
"you want a standard C++ build system, you got one it's called CMake, resistance is futile just use CMake" -
Bryce Adelstein C++ Now Jul 2021



kitware

Allow the use of native compilers and systems

- Visual Studio
- XCode
- ninja
- make
- nmake
- …Something new….
- every compiler under the sun…



kitware

Allow the use of native compilers and systems

- Visual Studio
- XCode
- ninja
- make
- nmake
- …Something new….
- every compiler under the sun…

cross-platform systems, CMake is designed to be used in conjunction with the
directory (called CMakeLists.txt files) are used to generate standard build files
(e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are
used in the usual way. CMake can compile source code, create libraries,

Simplify the build process

2024 Annual C++ Developer Survey "Lite"

	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A SIGNIFICANT ISSUE FOR ME	TOTAL	WEIGHTED AVERAGE
Managing libraries my application depends on	45.43% 571	36.44% 458	18.14% 228	1,257	2.27
Build times	42.86% 537	37.35% 468	19.79% 248	1,253	2.23
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	30.35%	42.53%	27.12% 336	1,239	2.03
Managing CMake projects	30.38% 377	38.20% 474	31.43% 390	1,241	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	27.67% 347	41.87% 525	30.46% 382	1,254	1.97
Setting up a development environment from scratch (compiler, build system, IDE, ...)	26.27% 330	41.80% 525	31.93% 401	1,256	1.94
Debugging memory problems	18.77% 234	49.24% 614	32.00% 399	1,247	1.87
Parallelism support: Using more CPU/GPU/other cores to compute an answer faster	22.94% 286	36.09% 450	40.98% 511	1,247	1.82
Memory safety: Bounds safety issues (read/write beyond the bounds of an object or array)	20.48% 257	35.86% 450	43.67% 548	1,255	1.77
Memory safety: Use-after-delete/free (dangling pointers, iterators, spans, ...)	20.03% 251	34.00% 426	45.97% 576	1,253	1.74
Managing Makefiles	19.88% 235	22.42% 265	57.70% 682	1,182	1.62

kitware

Simplify the build process

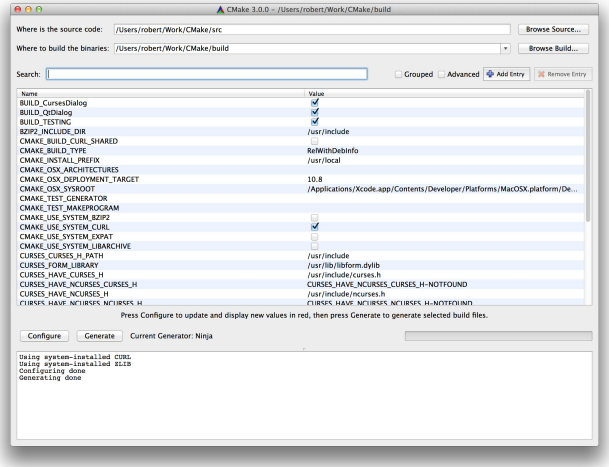
2024 Annual C++ Developer Survey "Lite"

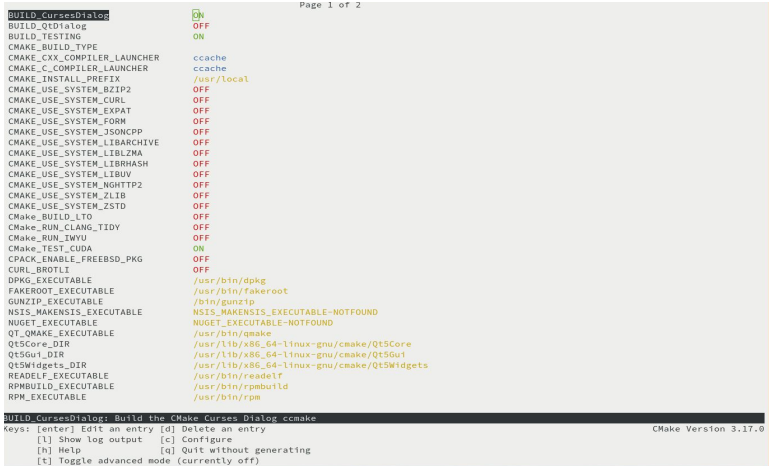
	MAJOR PAIN POINT	MINOR PAIN POINT	NOT A SIGNIFICANT ISSUE FOR ME	TOTAL	WEIGHTED AVERAGE
Managing libraries my application depends on	45.43% 571	36.44% 458	18.14% 228	1,257	2.27
Build times	42.86% 537	37.35% 468	19.79% 248	1,253	2.23
Setting up a continuous integration pipeline from scratch (automated builds, tests, ...)	30.35%	42.53%	27.12% 336	1,239	2.03
Managing CMake projects	30.38% 377	38.20% 474	31.43% 390	1,241	1.99
Concurrency safety: Races, deadlocks, performance bottlenecks	27.67% 347	41.87% 525	30.46% 382	1,254	1.97
Setting up a development environment from scratch (compiler, build system, IDE, ...)	26.27% 330	41.80% 525	31.93% 401	1,256	1.94
Debugging (memory issues, ...)	18.77% 234	49.24% 614	32.00% 399	1,247	1.87
Parallelism support: Using more CPU/GPU/other cores to compute an answer faster	22.94% 286	36.09% 450	40.98% 511	1,247	1.82
Memory safety: Bounds safety issues (read/write beyond the bounds of an object or array)	20.48% 257	35.86% 450	43.67% 548	1,255	1.77
Memory safety: Use-after-delete/free (dangling pointers, iterators, spans, ...)	20.03% 251	34.00% 426	45.97% 576	1,253	1.74
Managing Makefiles	19.88% 235	22.42% 265	57.70% 682	1,182	1.62

kitware

Optionally provide a user-interface to manage the build system

- cmake (non-interactive command line)
- ccmake (the terminal cli)
- cmake-gui (the Qt gui)





kitware

Optionally provide a user-interface to manage the build system

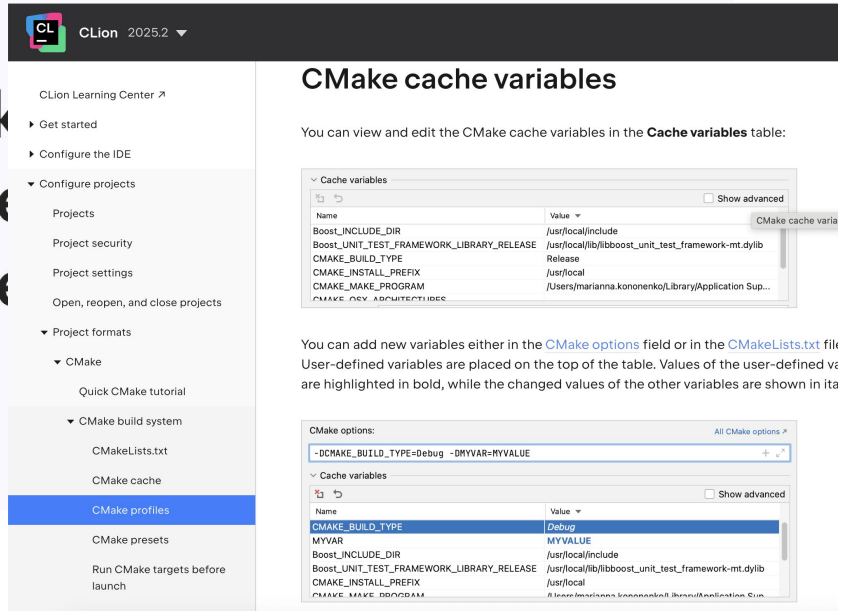
Many IDEs and Editors also provide GUIs for CMake

Dev Blogs > C++ Team Blog > Visual Studio Code CMake Tools Extension 1.17 Update: CMake Presets v6, Overriding Cache Variables, and Side Bar Updates

February 13th, 2024 0 reactions

Visual Studio Code CMak
1.17 Update: CMake Prese
Cache Variables, and Side

Sinem Akinci
Product Manager II



kitware

Optionally provide a user-interface to manage the build system

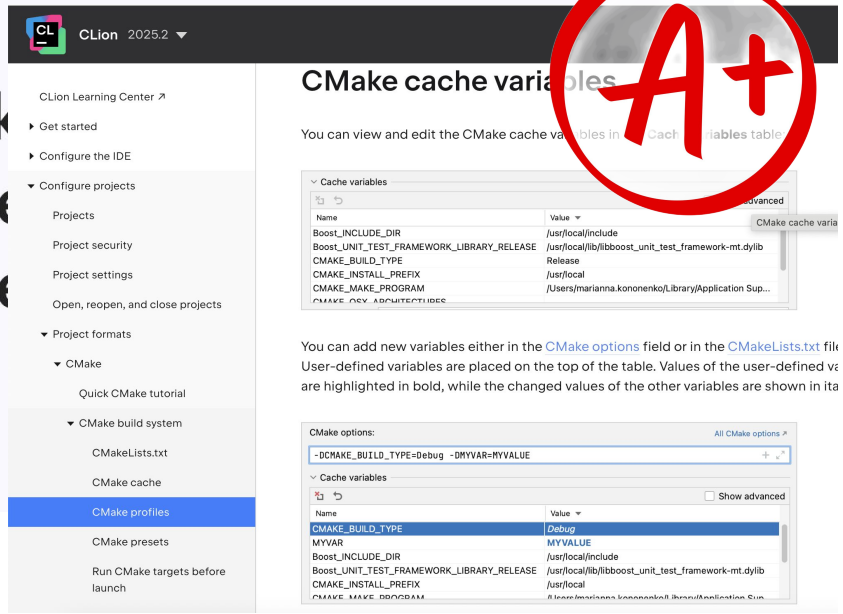
Many IDEs and Editors also provide GUIs for CMake

Dev Blogs > C++ Team Blog > Visual Studio Code CMake Tools Extension 1.17 Update: CMake Presets v6, Overriding Cache Variables, and Side Bar Updates

February 13th, 2024 0 reactions

Visual Studio Code CMak
1.17 Update: CMake Prese
Cache Variables, and Side

Sinem Akinci
Product Manager II

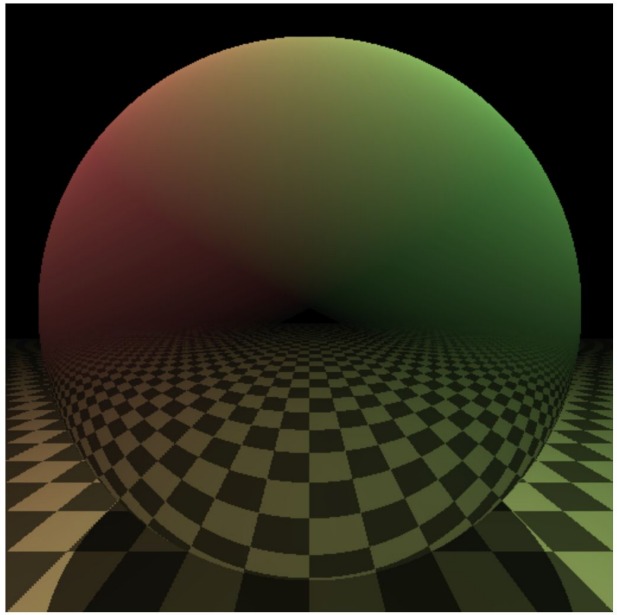


kitware

Create an extensible framework

CMake Ray Tracer

A simple ray tracer written in pure CMake. Inspired by raytracer.hpp. More information can be found at my blog.



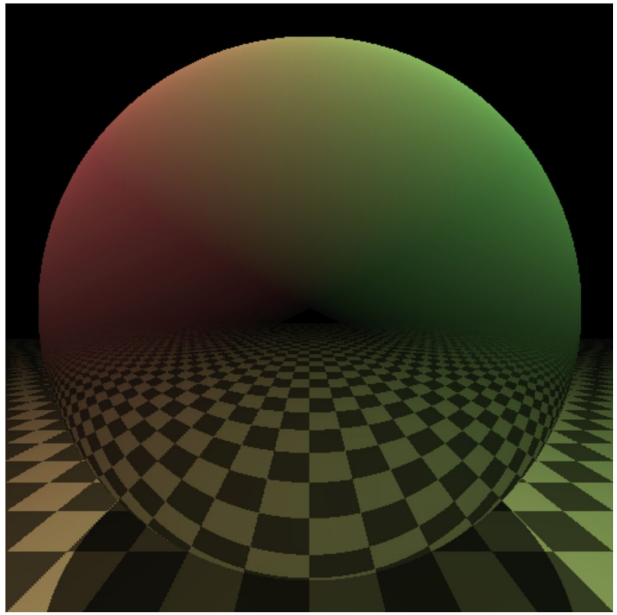
The CMake Language is no ones favorite part of CMake

kitware

Create an extensible framework

CMake Ray Tracer

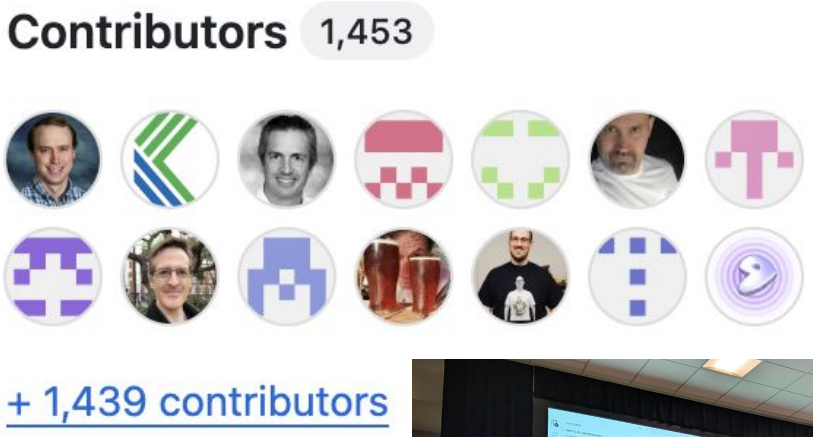
A simple ray tracer written in pure CMake. Inspired by raytracer.hpp. More information can be found at my blog.

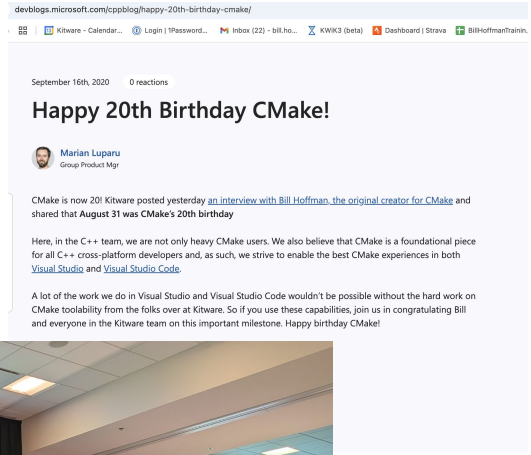


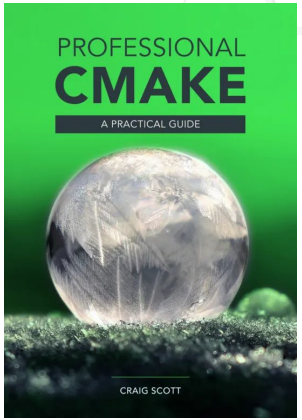
The CMake Language is no ones favorite part of CMake

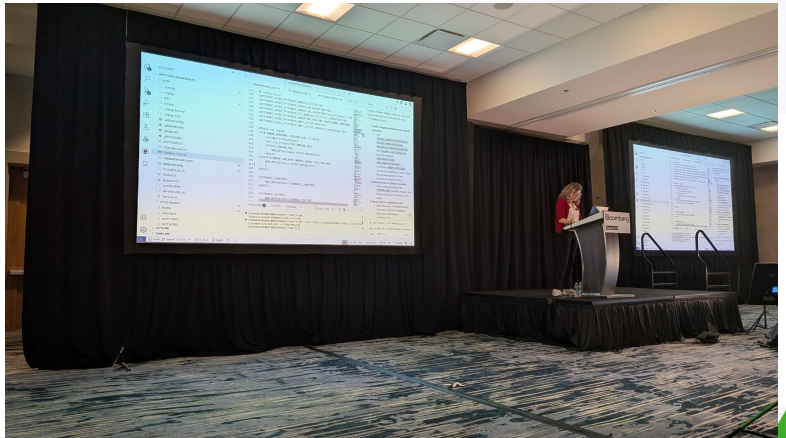
kitware

Self-sustaining community of software users and developers



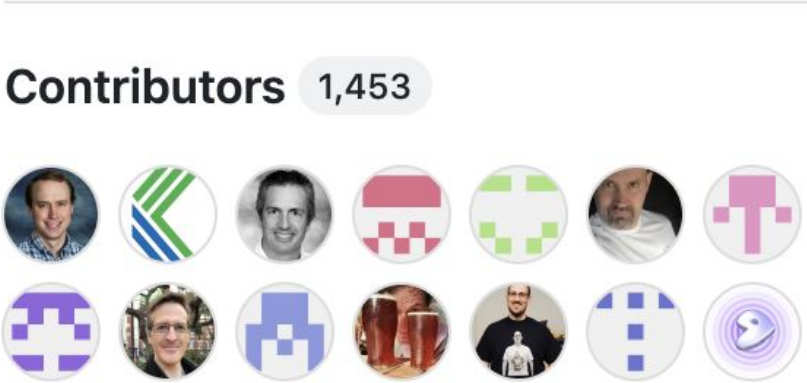


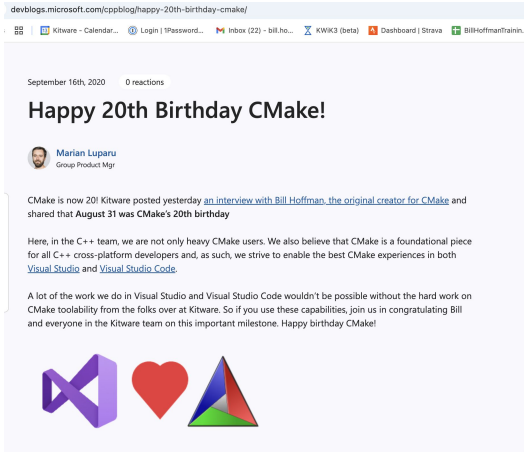


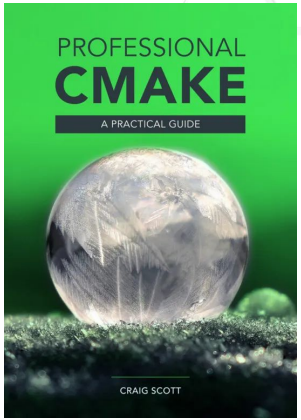


kitware

Self-sustaining community of software users and developers





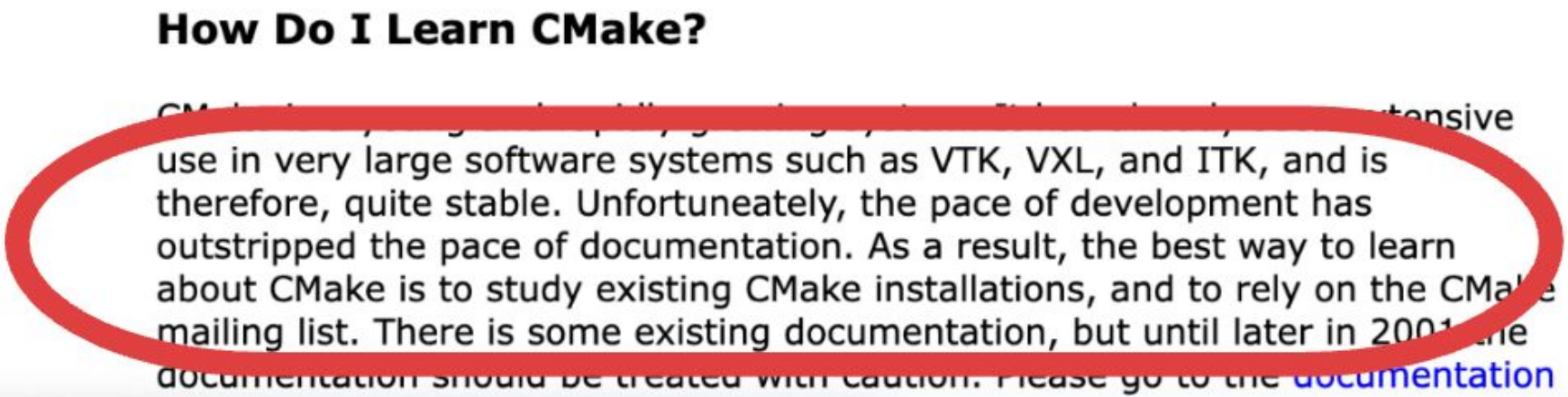


kitware

CMake 4

- **Implementing/defining formats that other applications can use**
- **CPS is json and will be available to IDEs, package managers and other tools to consume**
- **Modules standard files**
  - p1689 - scanner format
  - p3286 - module metadata format for import std (will be used by CPS)

Old CMake in the wild

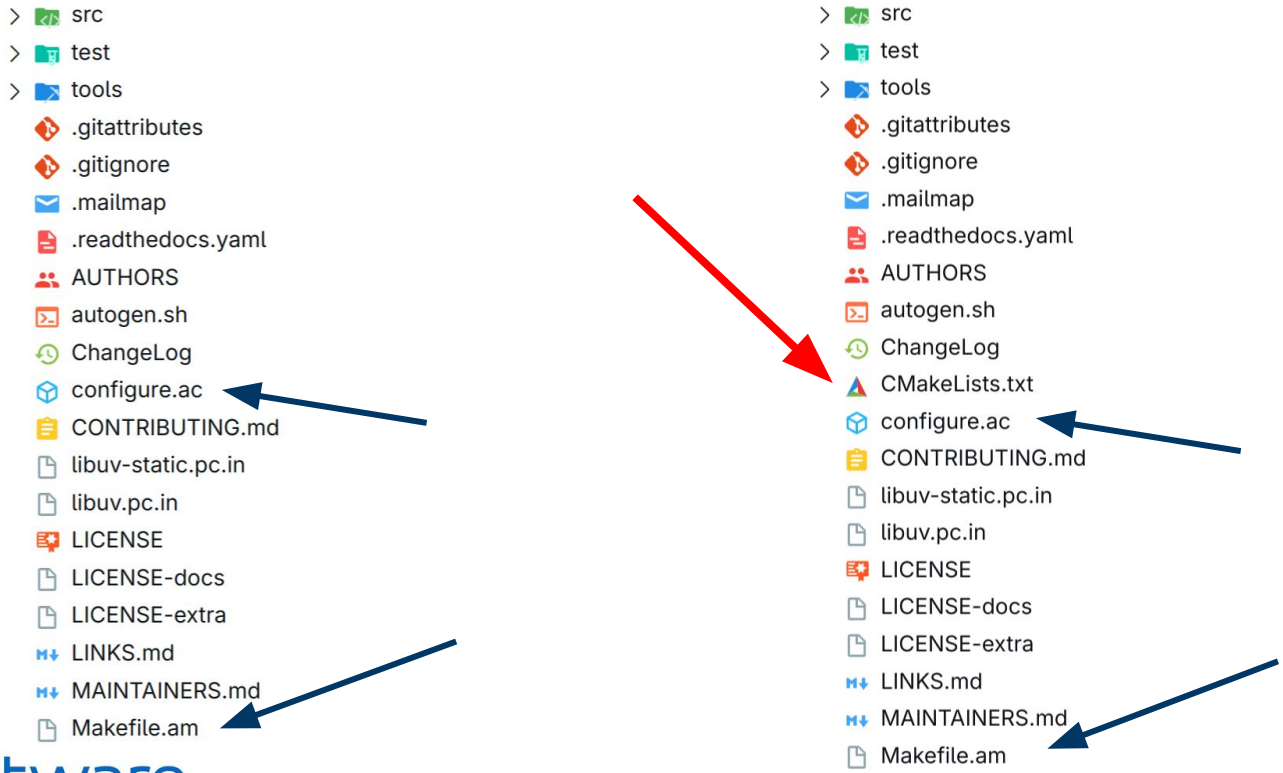


How Do I Learn CMake?

[illegible]tensive use in very large software systems such as VTK, VXL, and ITK, and is therefore, quite stable. Unfortuneately, the pace of development has outstripped the pace of documentation. As a result, the best way to learn about CMake is to study existing CMake installations, and to rely on the CMa[illegible]e mailing list. There is some existing documentation, but until later in 200[illegible]he documentation should be treated with caution. Please go to the documentation

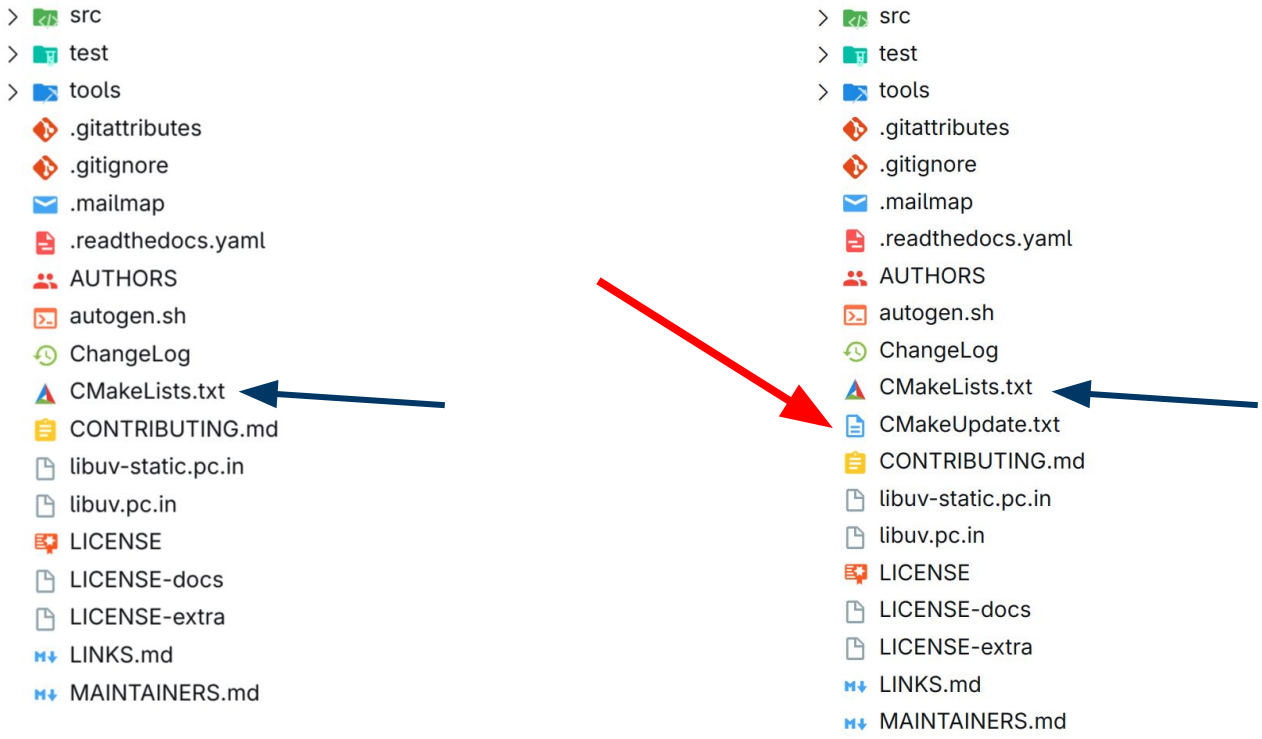
kitware

New in CMake: Alternate Filename for CMakeLists.txt



kitware

New in CMake: Alternate Filename for CMakeLists.txt



kitware

New in CMake: Alternate Filename for CMakeLists.txt

- `cmake --project-file <project-file-name>`
- Works with `add_subdirectory()`
- Uses CMakeLists.txt as a fallback if the new project file can’t be found
- Intended to be temporary, *always emits a warning*

What's cooking in CMake?

https://gitlab.kitware.com/cmake/cmake/-/blob/master/Help/dev/experimental.rst

master cmake / Help / dev / experimental.rst

experimental.rst

Experimental: Update the Instrumentation UUID
Tyler Yankee authored 1 week ago

experimental.rst 5.18 KiB

CMake Experimental Features Guide

The following is a guide to CMake experimental features that are under development and not yet included in official documenta
Development for more information.

Features are gated behind `CMAKE_EXPERIMENTAL_` variables which must be set to specific values in order to enable their gated f
change over time to reinforce their experimental nature. When used, a warning will be generated to indicate that an experiment
behavior in the project is not part of CMake's stability guarantees.

Export Package Dependencies

In order to activate support for this experimental feature, set

- variable `CMAKE_EXPERIMENTAL_EXPORT_PACKAGE_DEPENDENCIES` to
- value `1942b4fa-b2c5-4546-9385-83f254070067`.

This UUID may change in future versions of CMake. Be sure to use the value documented here by the source tree of the versio

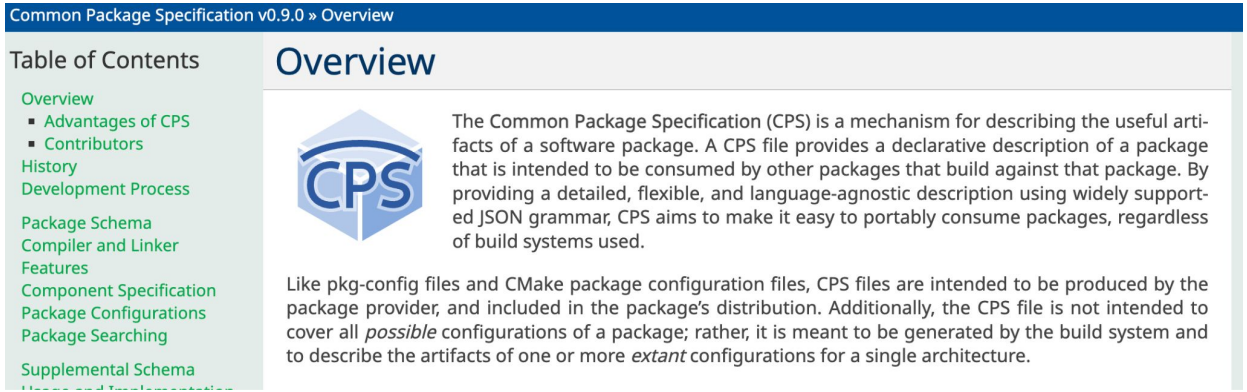
When activated, this experimental feature provides the following:

kitware

Current Experiments in CMake

- **Export Package Dependencies**
  - The install(EXPORT) and export(EXPORT) commands have experimental EXPORT_PACKAGE_DEPENDENCIES arguments to generate find_dependency calls automatically.
  - Details of the calls may be configured using the export(SETUP) command's PACKAGE_DEPENDENCY argument.
  - The package name associated with specific targets may be specified using the CMAKE_EXPORT_FIND_PACKAGE_NAME variable and/or
- **Export Common Package Specification Package Information**
  - The experimental install(PACKAGE_INFO) command is available to export package information in the Common Package Specification format.
- **Find/Import Common Package Specification Packages**
  - The :command:`find_package` command will also search for packages which are described using Common Package Specification. Refer to the :command:`find_package` documentation for details.
- **C++ import std support**
- **Build database support - json file with build information for building modules**
- **Instrumentation - build / test instrumentation**

CPS is here at last!



Common Package Specification v0.9.0 » Overview

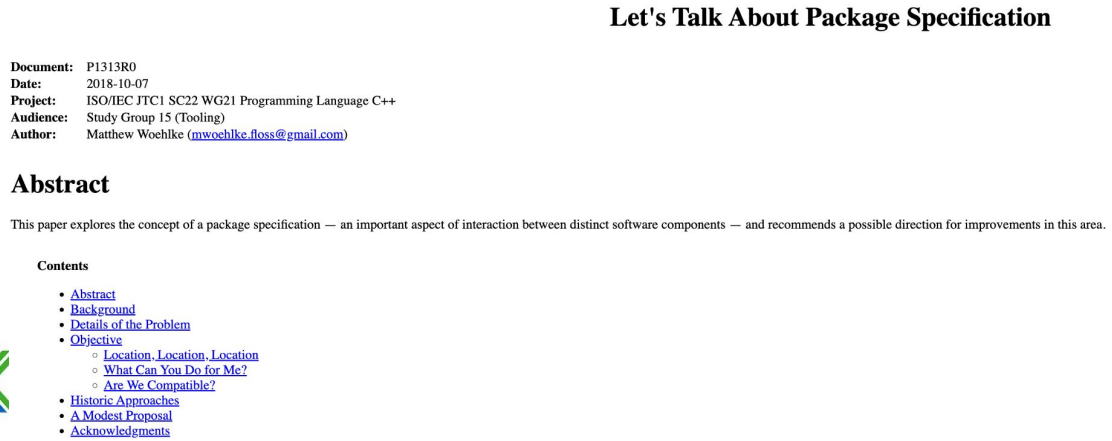
Table of Contents

- Overview
  - Advantages of CPS
  - Contributors
- History
- Development Process
- Package Schema
- Compiler and Linker Features
- Component Specification
- Package Configurations
- Package Searching
- Supplemental Schema

Overview

The Common Package Specification (CPS) is a mechanism for describing the useful artifacts of a software package. A CPS file provides a declarative description of a package that is intended to be consumed by other packages that build against that package. By providing a detailed, flexible, and language-agnostic description using widely supported JSON grammar, CPS aims to make it easy to portably consume packages, regardless of build systems used.

Like pkg-config files and CMake package configuration files, CPS files are intended to be produced by the package provider, and included in the package's distribution. Additionally, the CPS file is not intended to cover all *possible* configurations of a package; rather, it is meant to be generated by the build system and to describe the artifacts of one or more *extant* configurations for a single architecture.



Let's Talk About Package Specification

Document: P1313R0
Date: 2018-10-07
Project: ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience: Study Group 15 (Tooling)
Author: Matthew Woehlke (mwoehlke.floss@gmail.com)

Abstract

This paper explores the concept of a package specification — an important aspect of interaction between distinct software components — and recommends a possible direction for improvements in this area.

Contents

- Abstract
- Background
- Details of the Problem
- Objective
  - Location, Location, Location
  - What Can You Do for Me?
  - Are We Compatible?
- Historic Approaches
- A Modest Proposal
- Acknowledgments

- Started in 2016 by Matthew Woehlke
- 2018: WG21 paper
- 2022: C++Now talk by Bret and Daniel
- 2022 CppCon hallway track with Conan, vcpkg and others
- 2023: 3 talks at CppCon
- 2024: working on export in CMake
- 2025 C++ now talk

Common Package Specification

- Language & Platform agnostic package discovery mechanism
- Tool agnostic (JSON!), portable beyond the CMake ecosystem
- Open source and publicly developed, learn more at: github.com/cps-org/cps

Common Package Specification

- Multiple configurations for a given package
- Multiple components
- Flexible version compatibility and verification
- Relocatable by default
- Transitive dependencies “Just Work”

```
{
  "name": "sample",
  "description": "Sample CPS",
  "license": "BSD",
  "version": "1.2.0",
  "compat_version": "0.8.0",
  "platform": {
    "isa": "x86_64",
    "kernel": "linux",
    "c_runtime_vendor": "gnu",
    "c_runtime_version": "2.20",
    "jvm_version": "1.6"
  },
  "configurations": [ "optimized", "debug" ],
  "default_components": [ "sample" ],
  "components": {
    "sample-core": {
      "type": "interface",
      "definitions": [ "SAMPLE" ],
      "includes": [ "@prefix@/include" ]
    },
```

kitware

Common Package Specification

Old

```
install(
  EXPORT Example-targets
)

install(
  FILES
    Example-config.cmake
    Example-config-version.cmake
)
```

New

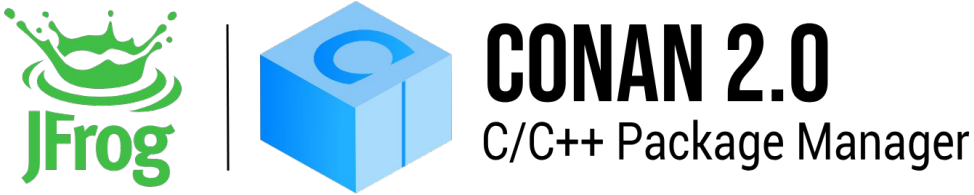
```
install(
  PACKAGE_INFO Example
  EXPORT Example-targets
)
```

Same

```
find_package(Example)
```

kitware

CPS and package managers



Spack

kitware

How to Contribute

CPS reference docs: https://cps-org.github.io/cps/

CPS project: https://github.com/cps-org/cps

Slack: cpplang.slack.com #ecosystem_evolution

Mailing list:
https://groups.google.com/g/cxx-ecosystem-evolution/about

Ecosystem and ISO discussions: https://github.com/isocpp/pkg-fmt/

kitware

CMake C++ module support

- C++20 named module
- C++ import std

kitware

C++ Modules Simple Example (build order order matters)

```
B.cpp:

export module B;
export void b() { }
```

```
A.cpp:

export module A;
import B;
export void a(){ b();}
```

```
 cl -std:c++20  -interface -c A.cpp

A.cpp
A.cpp(2): error C2230: could not find module 'B'
A.cpp(3): error C3861: 'b': identifier not found
```

kitware

Simple Example (build order matters)

B.cpp:

```
export module B;
export void b() { }
```

A.cpp:

```
export module A;
import B;
export void a(){ b();}
```

```
cl -std:c++20  -interface -c B.cpp
B.cpp

cl -std:c++20  -interface -c A.cpp
A.cpp
$ ls
A.cpp  A.ifc  A.obj  B.cpp  B.ifc  B.obj
```

kitware

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1689r5.html

defining a format for compilers p1689

Format for describing dependencies of source files

Ben Boeckel, Brad King
<ben.boeckel@kitware.com, brad.king@kitware.com>
version P1689R5, 2022-06-03

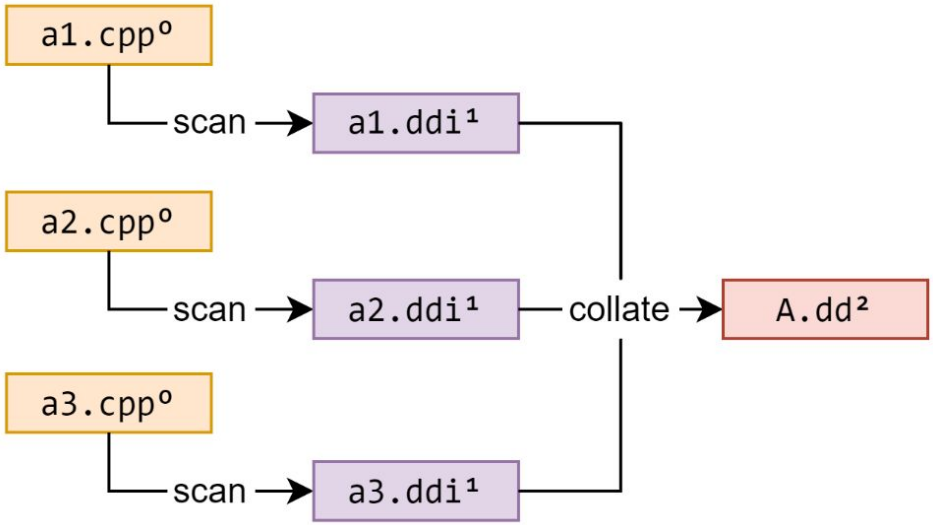
Table of Contents

Document number	ISO/IEC/JTC1/SC22/WG21/P1689R5
Date	2022-06-03
Reply-to	Ben Boeckel, Brad King, ben.boeckel@kitware.com, brad.king@kitware.com
Audience	EWG (Evolution), SG15 (Tooling)

kitware

C++20 Modules

- Named Module Support has arrived
- Produce and consume modules within a project
- Export Modules using standard CMake install() commands
- Discover modules installed by other packages



kitware

C++20 Modules

```
add_library(modlib)

target_sources(modlib PUBLIC
  FILE_SET CXX_MODULES
  FILES lib.cpp util.cpp
)
```

```
install(
  TARGETS modlib
  EXPORT modlib-targets
  FILE_SET CXX_MODULES
)

install(
  EXPORT modlib-targets
  NAMESPACE modlib::
  CXX_MODULES_DIRECTORY modules
)
```

kitware

import std;

- It's almost here:
  - MSVC 14.35+
  - LLVM 19+
  - GCC 15+
- Faster builds
- Never worry about IWYU again

```
import std;

int main() {
 std::println("Hello import std");
}
```

kitware

import std;

```
add_executable(hello
  hello.cpp
)

target_compile_features(hello
  PRIVATE cxx_std_23
)
```

```
set(CMAKE_CXX_MODULE_STD 1)

#or

set_target_properties(hello
  PROPERTIES CXX_MODULE_STD 1
)
```

kitware

import std (generated library)

```
[1/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=g
[2/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../li
[3/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=g
[4/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/main.dir/CXXDependInfo.json --lang=C
[5/14] "CLANG_ROOT/bin/clang-scan-deps" -format=p1689 -- CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../li
[6/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/__cmake_cxx23.dir/CXXDependInfo.jso
[7/14] CMAKE_ROOT/bin/cmake -E cmake_ninja_dyndep --tdi=CMakeFiles/uses_std.dir/CXXDependInfo.json --la
[8/14] CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=gnu++20 -MD -MT CMakeFiles/main.dir/main.cxx.o -MF
[9/14] CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../lib/x86_64-unknown-linux-gnu/../../share/libc++/v1 -
[10/14] CLANG_ROOT/bin/clang++  -ICLANG_ROOT/bin/../lib/x86_64-unknown-linux-gnu/../../share/libc++/v1
[11/14] : && CMAKE_ROOT/bin/cmake -E rm -f lib__cmake_cxx23.a && CLANG_ROOT/bin/llvm-ar qc lib__cmake_c
[12/14] CLANG_ROOT/bin/clang++   -stdlib=libc++ -std=gnu++20 -MD -MT CMakeFiles/uses_std.dir/uses_std.c
[13/14] : && CMAKE_ROOT/bin/cmake -E rm -f libuses_std.a && CLANG_ROOT/bin/llvm-ar qc libuses_std.a  CM
[14/14] : && CLANG_ROOT/bin/clang++ -stdlib=libc++ -Wl,-rpath,CLANG_ROOT/lib/x86_64-unknown-linux-gnu C
```

kitware

CMake Instrumentation

- In CMake 4.0+, CMake can collect instrumentation data about each step of a project's build, test and install time.
  - Provide insight into performance over time across users and machines.
  - Generate Google Trace Event files for visualizing data to analyze parallelism and bottlenecks. (4.2+)

Visualizing a build using CMake Instrumentation and the Google Trace Event format

kitware

Snippet Files

- With instrumentation enabled, JSON files with instrumentation data are generated for different event types
  - configure
  - generate
  - compile
  - link
  - custom command
  - install
  - test
- Each snippet contains data such as:
  - timeStart
  - duration
  - role
  - outputSizes
  - CPU Load before/after
  - Host Memory usage before/after

Example snippet data for a compile command

```
"command"
    "/usr/bin/c++ -MD -MT
    CMakeFiles/main.dir/main.cxx.o -o
    CMakeFiles/main.dir/main.cxx.o -c main.cxx"
"role"   "compile"
"language"   "C++"
"outputs"
      "CMakeFiles/main.dir/main.cxx.o"
"outputSizes"     87664
"source"   "main.cxx"
"dynamicSystemInformation"

  "afterCPULoadAverage"   2.35
  "afterHostMemoryUsed"   6635680.0
  "beforeCPULoadAverage"   2.35
  "beforeHostMemoryUsed"   6635832.0

"timeStart"   1737053448177
"duration"   31
```

kitware

Indexing

- At user-defined intervals, CMake automatically collates all generated snippet files and creates an index file.
  - Example: After every build, after every configure, or when triggered manually with:

    ```
    ctest --collect-instrumentation
    ```

- Custom callback scripts will be run with the index file as an argument, after which the snippet files are cleaned up automatically.

Example index file

```
"hook"    "manual"
"buildDir"    "[build-dir]"
"dataDir"
  "[build-dir]/.cmake/instrumentation/v1/data"
"snippets"

  "compile-4b48bcd05bda-11-20T15-07-14-0741.json"
  "compile-7adad0ed5b82-11-20T16-17-40-0587.json"
  "configure-751fac5bfa96-11-20T16-17-38-0113.json"
  "custom-eca9f646550c-11-20T16-17-40-0387.json"
  "generate-751fac5bfa96-11-20T16-17-38-0134.json"
  "link-0db97bb54f2bd4-11-20T15-07-14-0755.json"
```

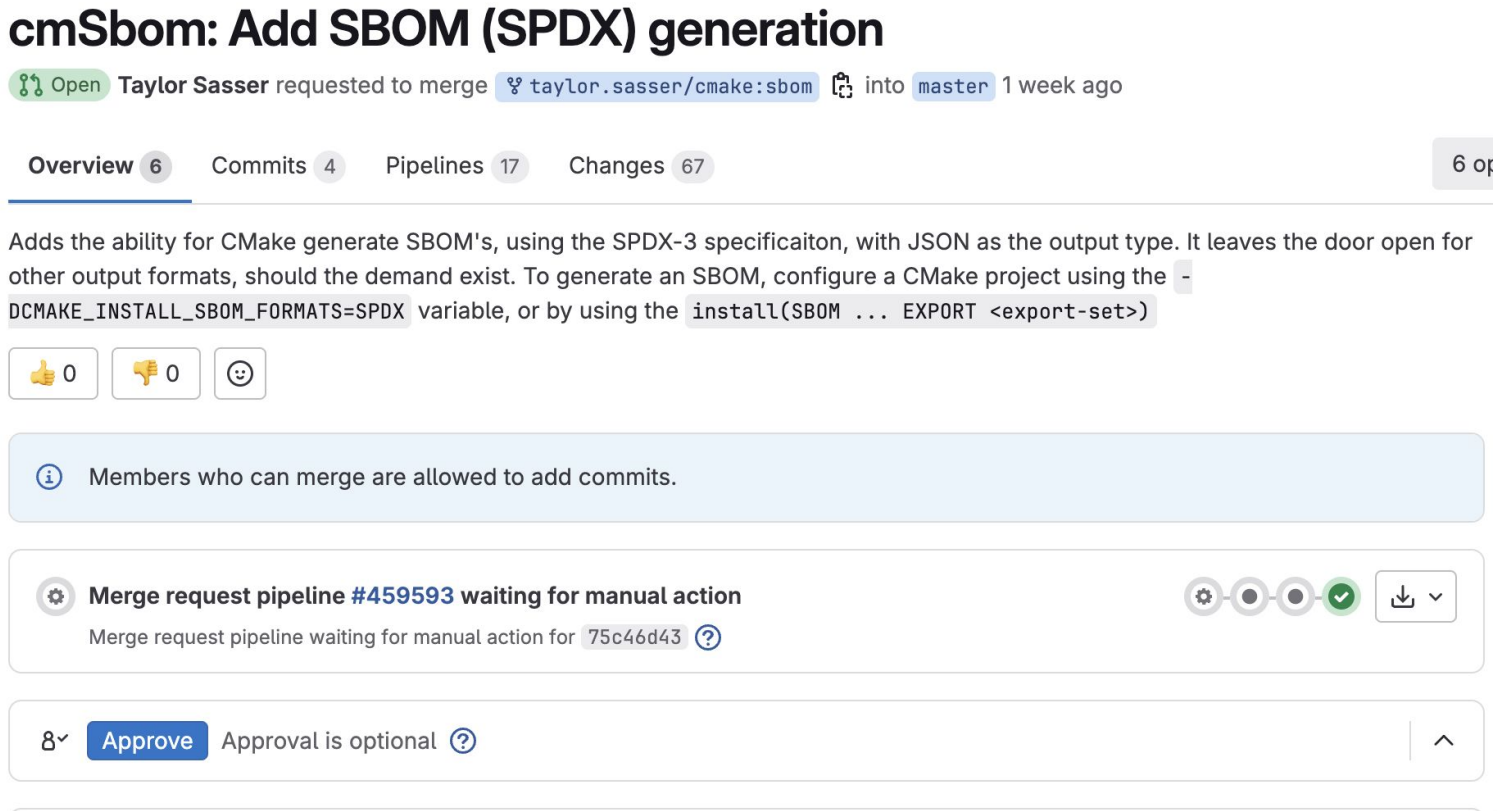
Enabling CMake Instrumentation

```
cmake_instrumentation(
  API_VERSION 1
  DATA_VERSION 1
  OPTIONS
     staticSystemInformation    # collect static system data when indexing
     dynamicSystemInformation   # collect dynamic system data before and after each command
     trace                      # automatically generate a Google Trace Event file
  HOOKS postGenerate postBuild postCTest # intervals at which to index data and run CALLBACK
  CALLBACK "${CMAKE_COMMAND} -P /path/to/handle_data.cmake # index file appended to args
)
```

Alternatively:

- A JSON file with similar options can be placed under `CMAKE_CONFIG_DIR` (ie `~/.config/cmake`) for user-wide instrumentation queries.
- Setting environment variable `CTEST_USE_INSTRUMENTATION=1` will enable instrumentation with a built-in callback to submit instrumentation data to CDash.

SBOM (almost in the kitchen)



SBOM CMake generation

```cmake
add_executable(application
  main.c
)

find_package(bar 1.3.4 CONFIG)
target_link_libraries(application PUBLIC bar::bar)

install(
  TARGETS application
  EXPORT application_targets
)

install(
  SBOM application_targets
  EXPORT application_targets
)
```

```cmake
# Alternatively, tell CMake to generate the SBOM via
# cmake -DCMAKE_INSTALL_SBOM_FORMATS=SPDX
add_executable(application
  main.c
)

find_package(bar 1.3.4 CONFIG)
target_link_libraries(application PUBLIC bar::bar)

install(
  TARGETS application
  EXPORT application_targets
)
```

SPDX output


```
{
  "@context" : "https://spdx.org/rdf/3.0.1/spdx-context.jsonld",
  "@graph" :
  [
    {
      "@id" : "_:application_targets#SPDXDocument",
      "element" :
      [
        {
          "@id" : "_:bar:bar#Package",
          "builtTime" : "2025-09-16T15:40:41Z",
          "copyrightText" : "MIT",
          "description" : "A description",
          "name" : "bar:bar",
          "originatedBy" :
          [
            {
              "name" : "bar",
              "type" : "Organization"
            }
          ],
          "packageUrl" : "https://github.com/Kitware/CMake/releases/download/v4.1.0/cmake-4.1.0.tar.gz",
          "packageVersion" : "1.3.5",
          "type" : "software_Package"
        }
      ],
      "name" : "application_targets",
      "profileConformance" :
      [
        "core",
        "software"
      ],
      "rootElement" :
      [
        {
          "@id" : "_:application#Package",
          "externalRef" :
          [
            {
              "comment" : "Build System used for this target",
              "externalRefType" : "buildSystem",
              "locator" : "_:CMake#Agent",
              "type" : "ExternalRef"
            }
          ],
          "name" : "application",
          "primaryPurpose" : "APPLICATION",
          "type" : "software_Package"
        }
      ],
      "type" : "SpdxDocument"
    },
    {
      "description" : "Linked Libraries",
      "from" : "_:application#Package",
      "relationshipType" : "DEPENDS_ON",
      "to" :
      [
        "_:bar:bar#Package"
      ],
      "type" : "Relationship"
    }
  ]
}
```


kitware

New Build Tool -GFASTBuild

FASTBUILD

Contact | License

Home | Features | Documentation | Download

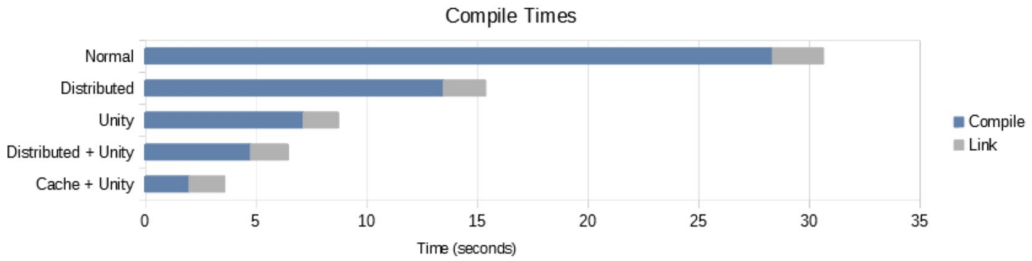
Welcome to FASTBuild!

FASTBuild is a high performance, open-source build system for Windows, Linux and OS X. It supports highly scalable compilation, caching and network distribution.

From the largest studios in the world to the smallest independent developers, FASTBuild is used in production every day to develop for PC/Mac/Linux, Consoles, Smartphones and retro systems.

FASTBuild gives you all the tools you need to speed up your builds, allowing for improvements of 10x or more over other build systems.

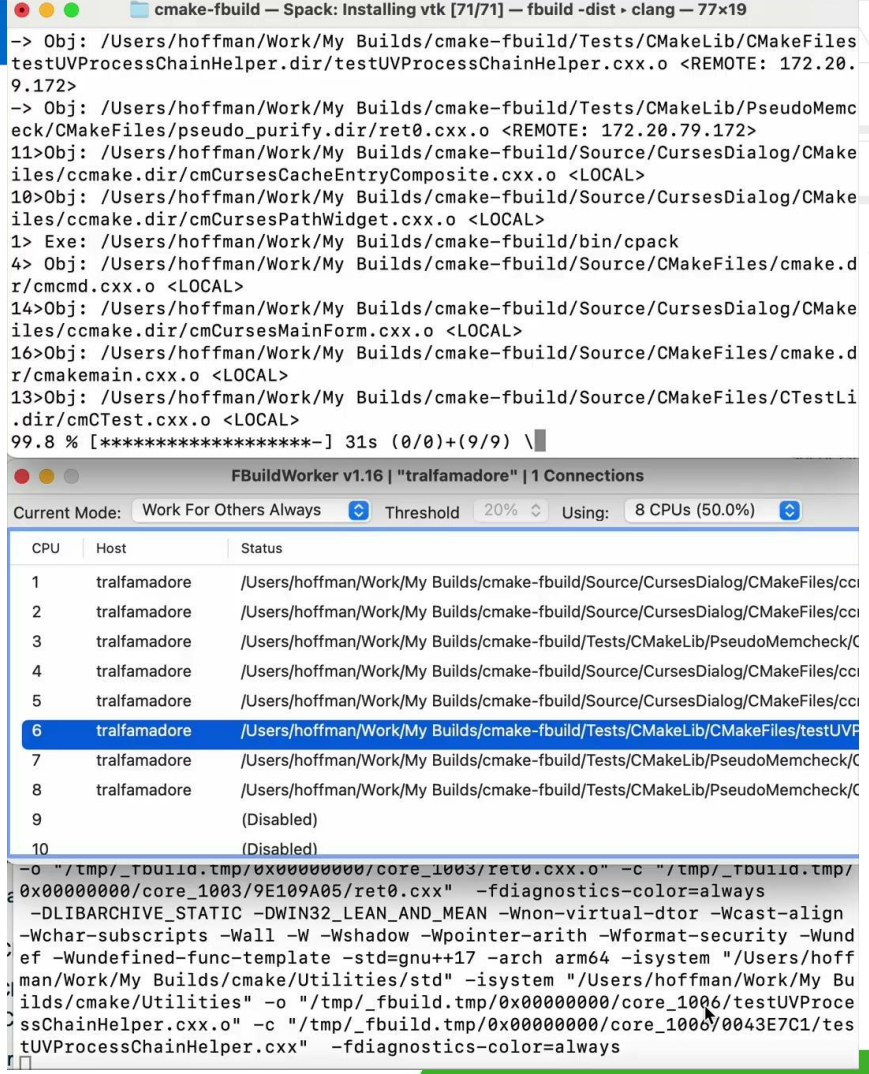
More info...



Thanks to: Eduard Voronkin

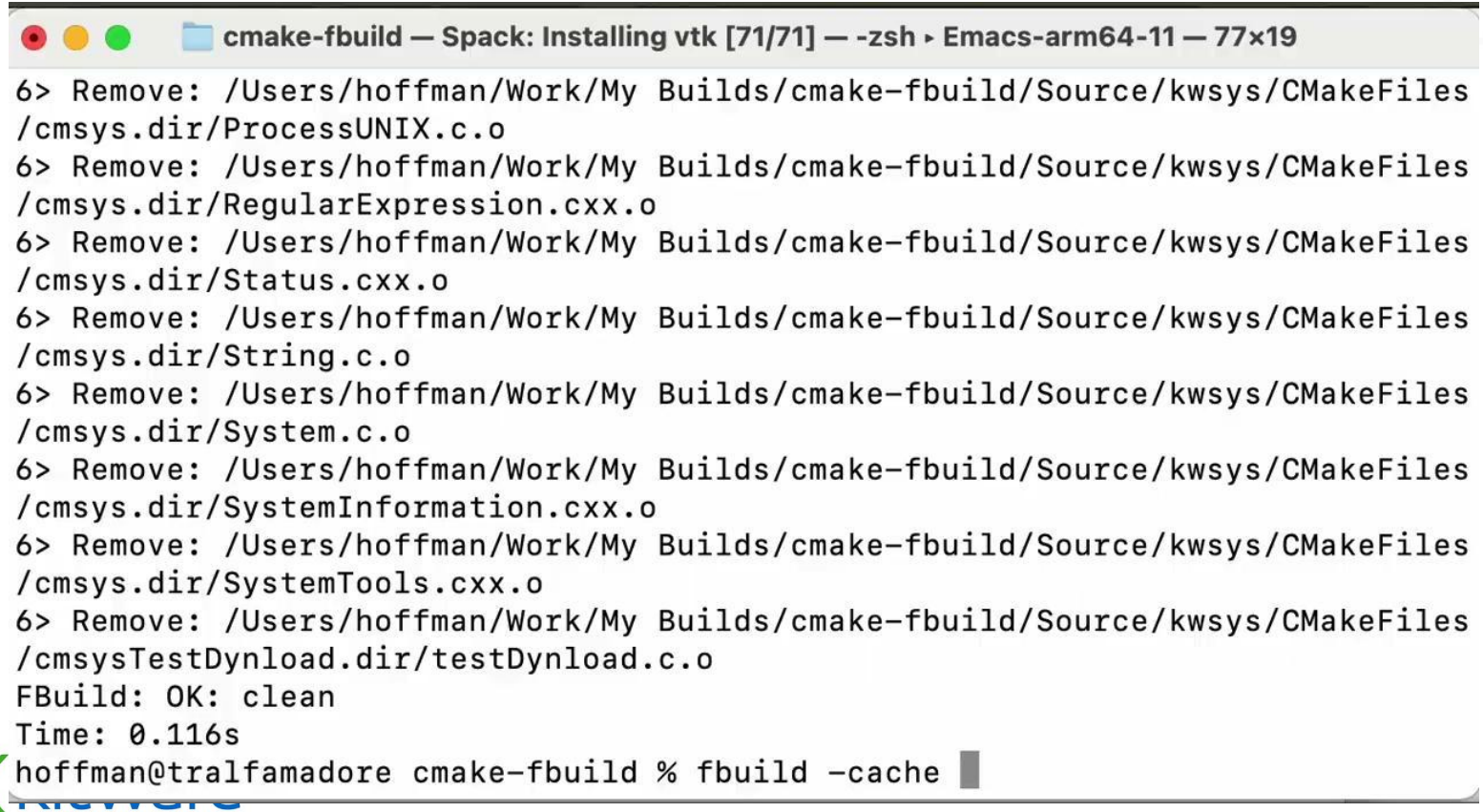
kitware

FASTBuild -dist

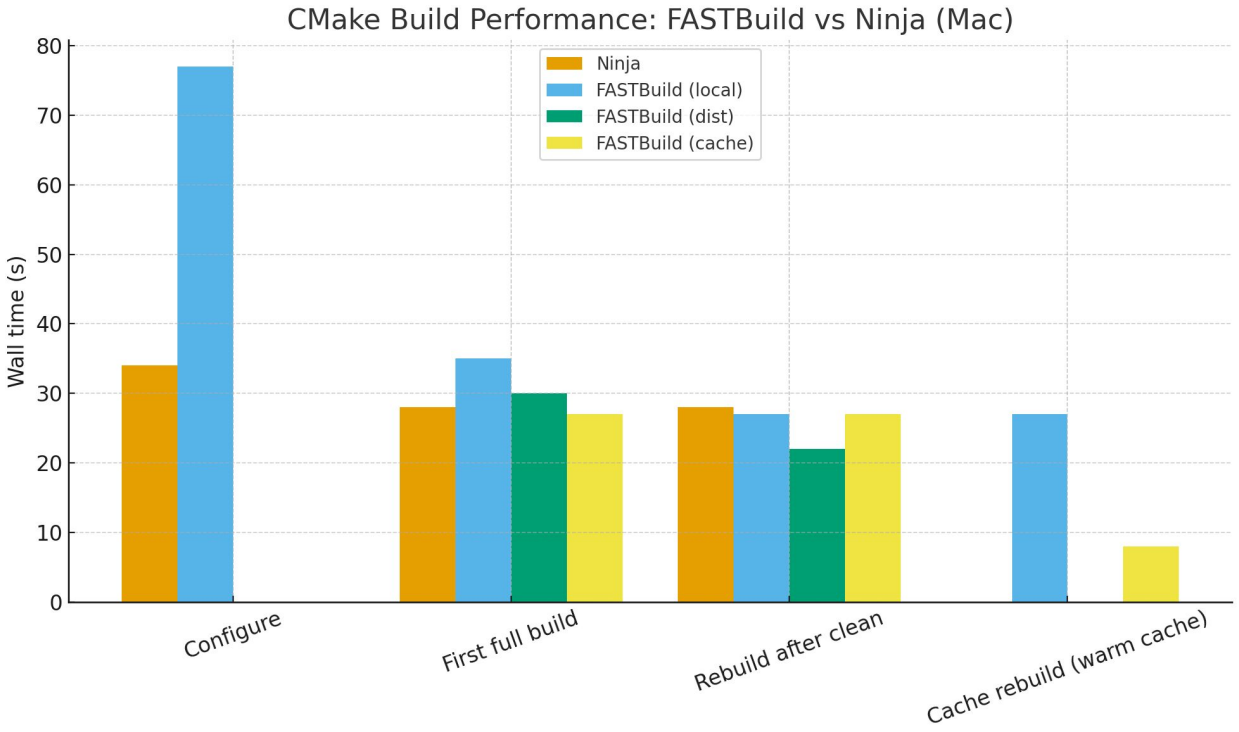


kitware

FASTBuild -cache

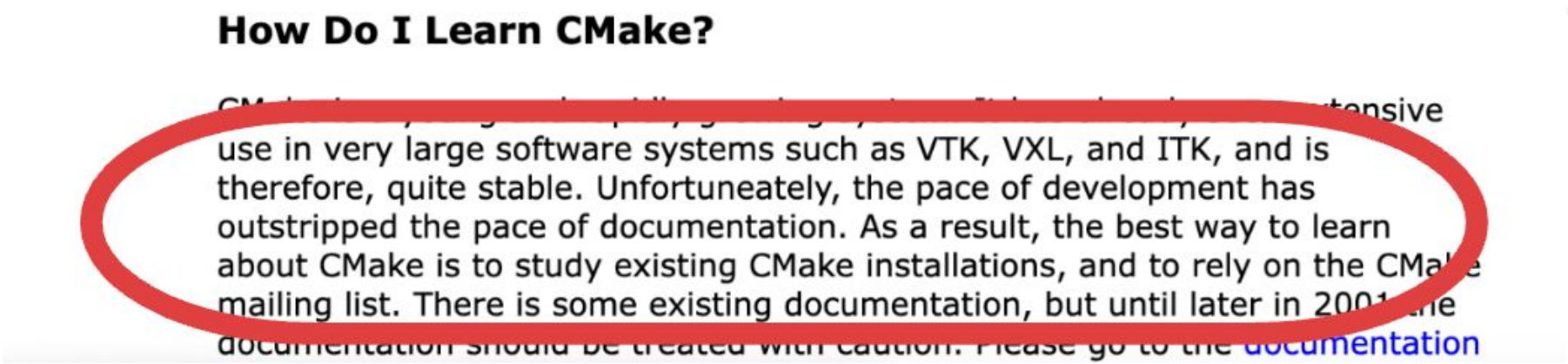


New Build Tool -GFASTBuild



kitware

New Modern Tutorial on the way CMake 4.2



How Do I Learn CMake?

...tensive use in very large software systems such as VTK, VXL, and ITK, and is therefore, quite stable. Unfortuneately, the pace of development has outstripped the pace of documentation. As a result, the best way to learn about CMake is to study existing CMake installations, and to rely on the CMa...e mailing list. There is some existing documentation, but until later in 200...he documentation should be treated with caution. Please go to the documentation

kitware

CMAKE_EXPORT_SARIF

Added in version 4.0.

Enable or disable CMake diagnostics output in SARIF format for a project.

If enabled, CMake will generate a SARIF log file containing diagnostic messages output by CMake when running in a project. By default, the log file is written to `.cmake/sarif/cmake.sarif`, but the location can be changed by setting the command-line option `cmake --sarif-output` to the desired path.

The Static Analysis Results Interchange Format (SARIF) is a JSON-based standard format for static analysis tools (including build tools like CMake) to record and communicate diagnostic messages. CMake generates a SARIF log entry for warnings and errors produced while running CMake on a project (e.g. `message()` calls). Each log entry includes the message, severity, and location information if available.

An example of CMake's SARIF output is:


```
{
  "version" : "2.1.0",
  "$schema" : "https://schemastore.azurewebsites.net/schemas/json/sarif-2.1.0-rtm.4.json",
  "runs" :
  [
    {
      "tool" :
      {
        "driver" :
        {
          "name" : "CMake",
          "rules" :
          [
            {
              "id" : "CMake.Warning",
              "messageStrings" :
              {
                "default" :
                {
                  "text" : "CMake Warning: {0}"
                }
              },
              "name" : "CMake Warning"
```

Future's so bright, gotta wear shades, Bill's dreams

kitware

CMake linting and easy detection of old cmake patterns

```
cmake_minimum_required(VERSION 2.8)
project(MyProject)

include_directories(${CMAKE_SOURCE_DIR}/include)
add_definitions(-DUSE_OLD)

add_executable(myapp main.cpp)
target_link_libraries(myapp ${CMAKE_THREAD_LIBS_INIT})
```

kitware

CMake linting and easy detection of old cmake patterns

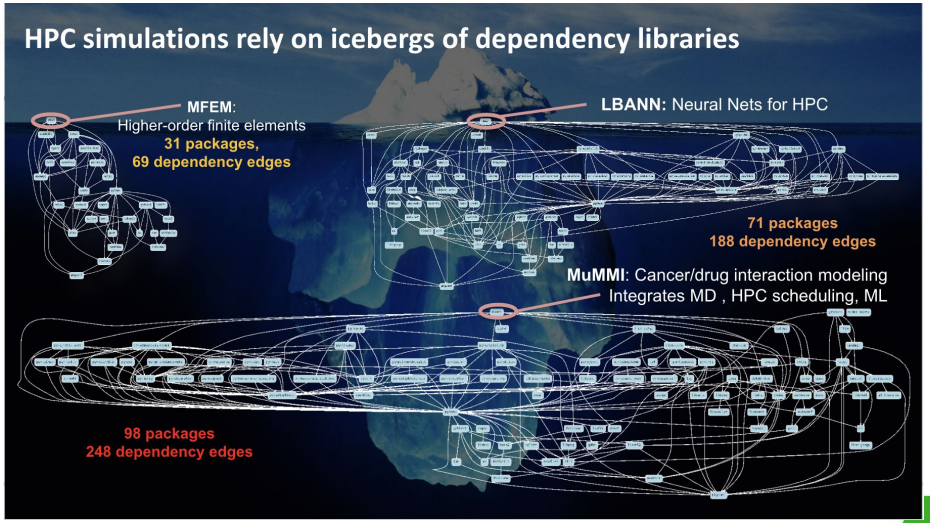
```
[WARNING] CMakeLists.txt:1
  Using 'cmake_minimum_required(VERSION 2.8)'.
  Suggest upgrading to at least 3.15 for modern features.

[STYLE] CMakeLists.txt:3
  Detected old-style 'include_directories()'.
  Use 'target_include_directories(myapp PRIVATE ...)' instead.

[STYLE] CMakeLists.txt:4
  'add_definitions()' found.
  Prefer 'target_compile_definitions(myapp PRIVATE ...)'.

[STYLE] CMakeLists.txt:6
  Linking with variable '${CMAKE_THREAD_LIBS_INIT}'.
  Prefer 'find_package(Threads)' + 'target_link_libraries(myapp Threads::Threads)'.
```

kitware

CMake is not a package manager

Turns out the build is not enough

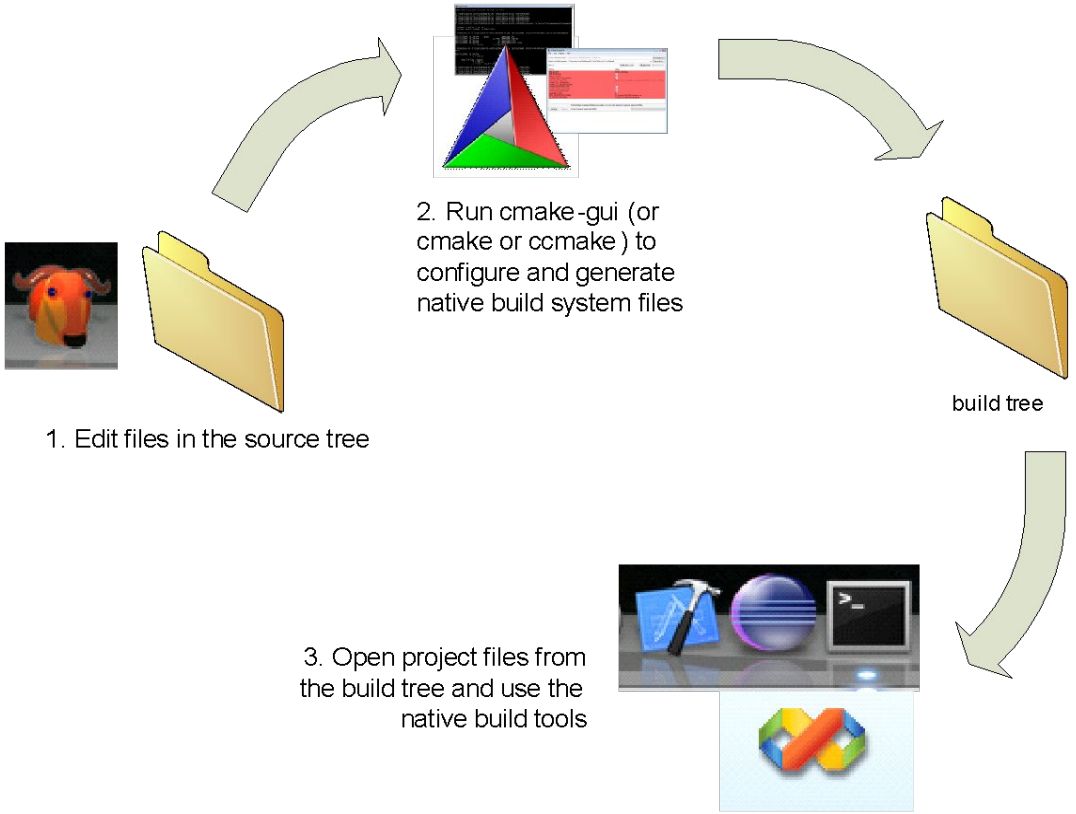
One of the main objectives of CMake was to not add external dependencies to the project. Since CMake was used to build C++, the only thing on which we could depend would be a valid C++ compiler. That is why CMake is written entirely in C/C++, so that it can be built on any system on which you need to build your C++ project. This avoids what I like to call the Easter egg hunt build process. First find/build/install these five things and then you are ready to build the project you actually want to use.



ExternalProject
FetchContent

kitware

CMake: Workflow

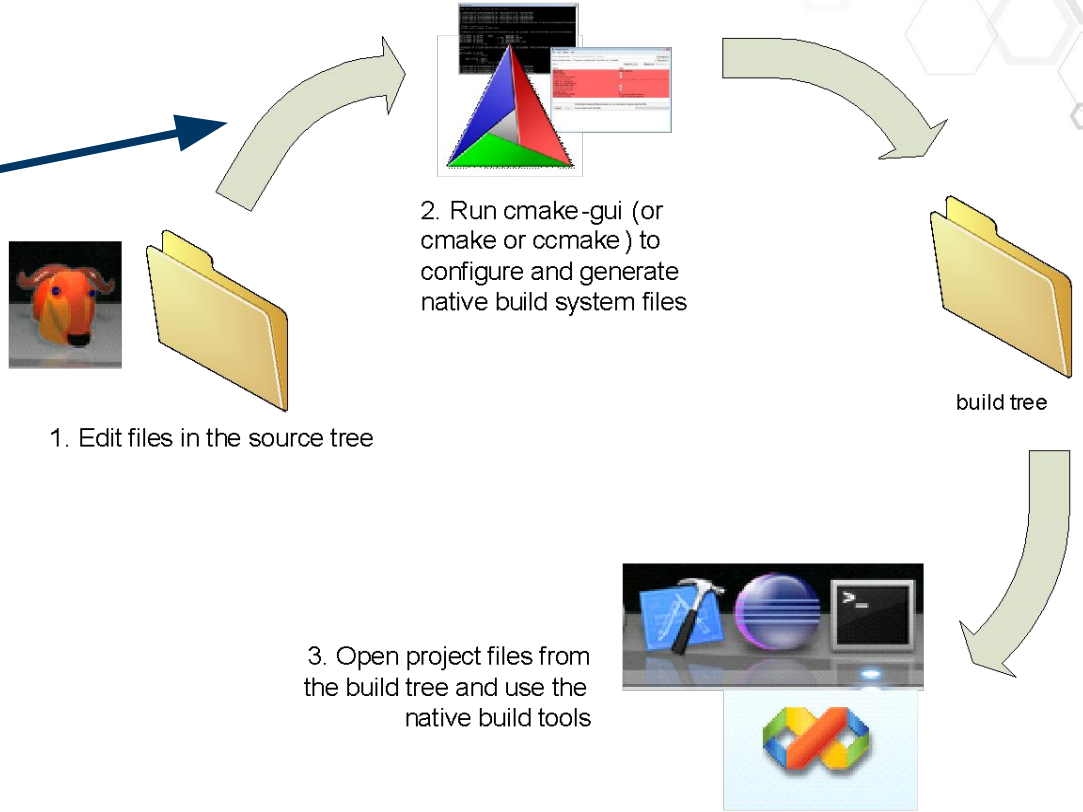


kitware

CMake: Workflow

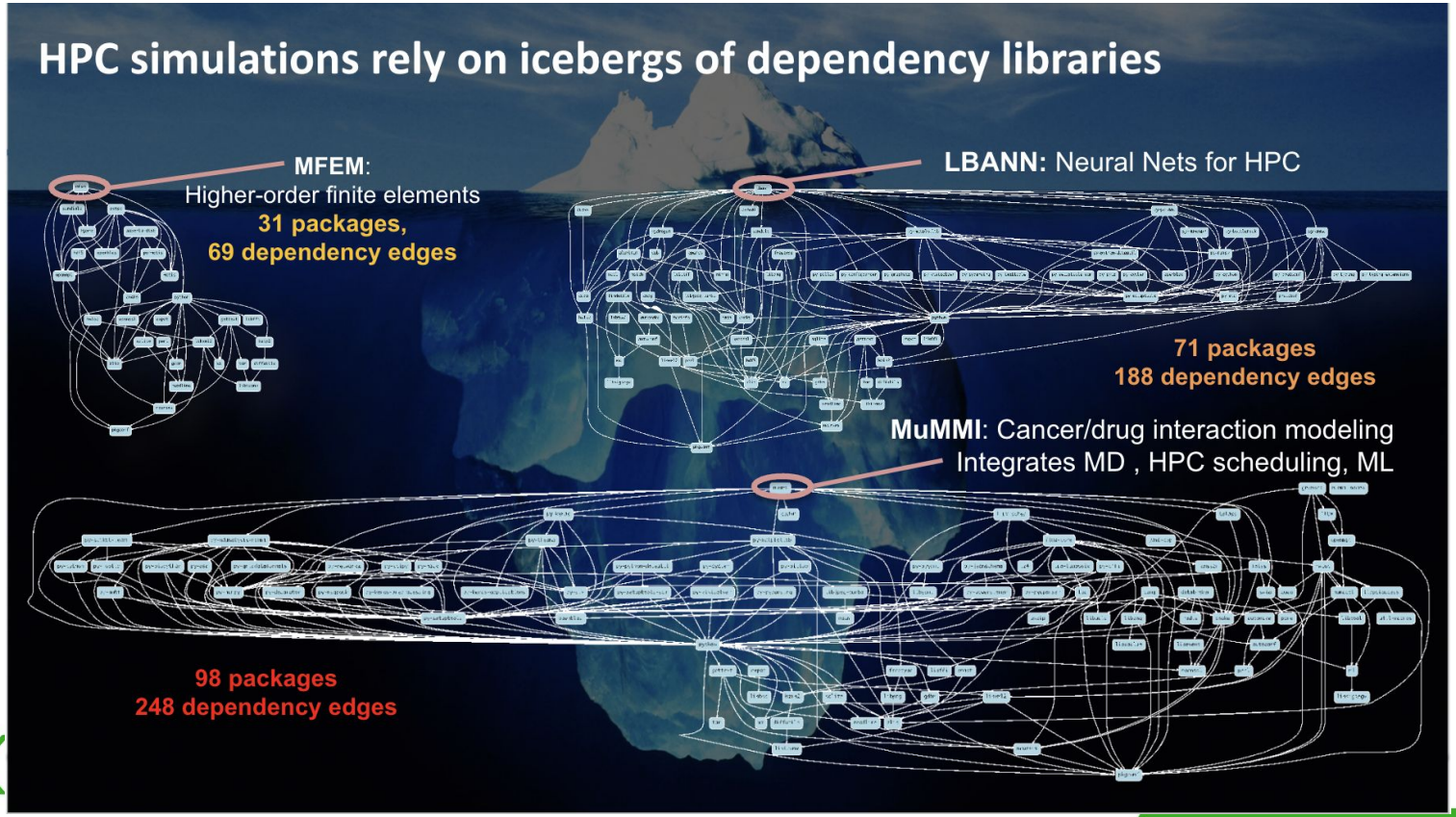
cmake provision

runs package manager of choice spack, conan, vcpkg, etc.



kitware

spack develop + CMake



spack develop + CMake

mypackage: libraries (A,B,C,D,E, F)

I want to develop A, B, and C.

Build or fetch binaries for all other depends

edit, compile, run, debug (A,B,C no install builds)

wait I want to debug F add it to develop

edit, compile, run, debug (A,B,C,F)

wait I only want to develop C

edit, compile, run, debug (C)

Eras of CMake

- CMake 1: Primordial Flag Soup (2000)
- CMake 2: World Domination (2004)
- CMake 3: I hope you like target_* commands (2014)
- CMake 4: interop and standardization I hope you like json

Thanks to all of you for helping to achieve this goal:

"Develop an open source, cross-platform tool to manage the build process"

kitware

Thanks and Join the community

https://gitlab.kitware.com/cmake/cmake/-/issues

https://discourse.cmake.org/

slack CppLang #cmake #ecosystem_evolution

www.kitware.com

kitware