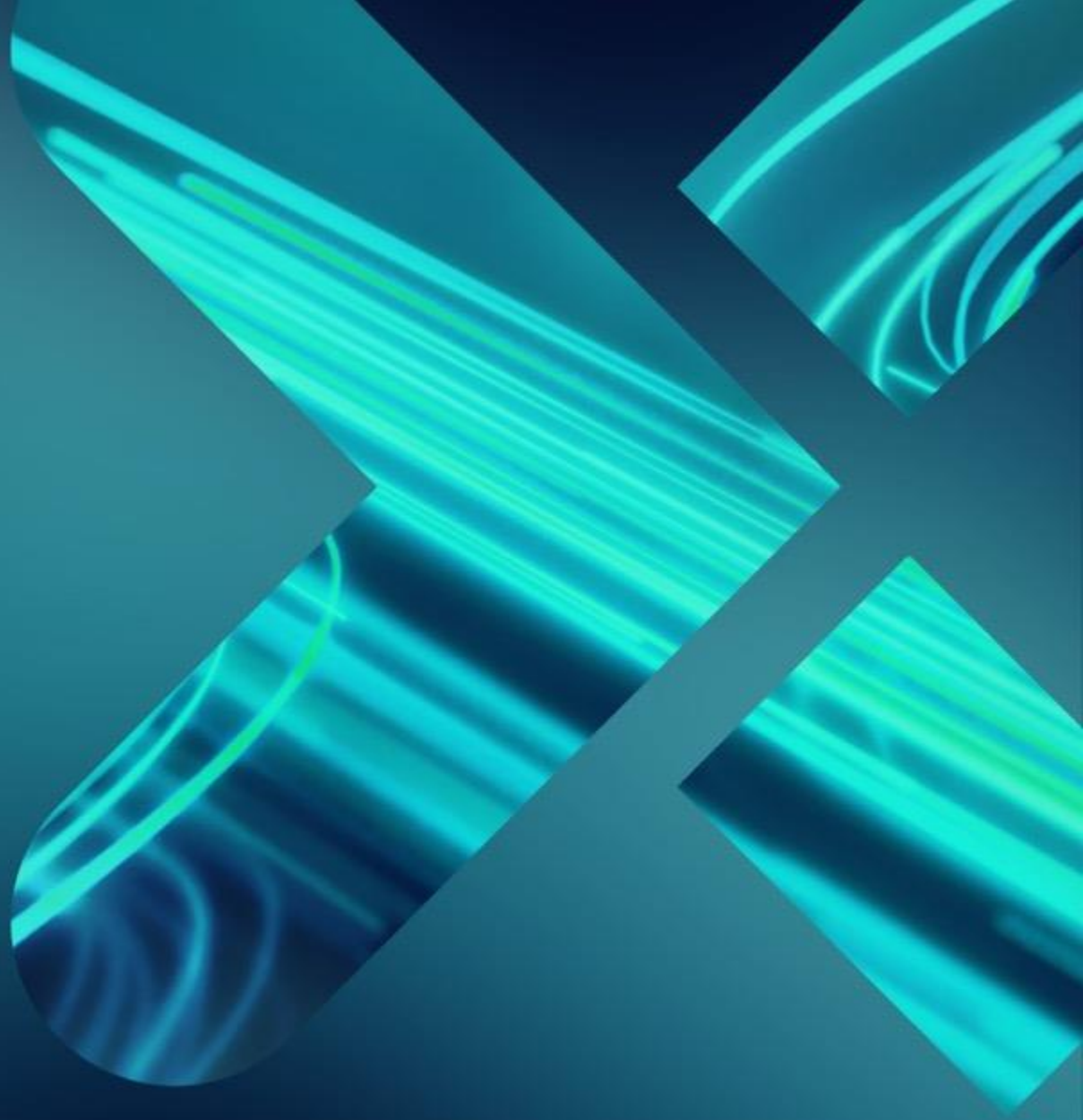
Effective clang-tidy

A case study in writing custom checks at scale



SIEMENS

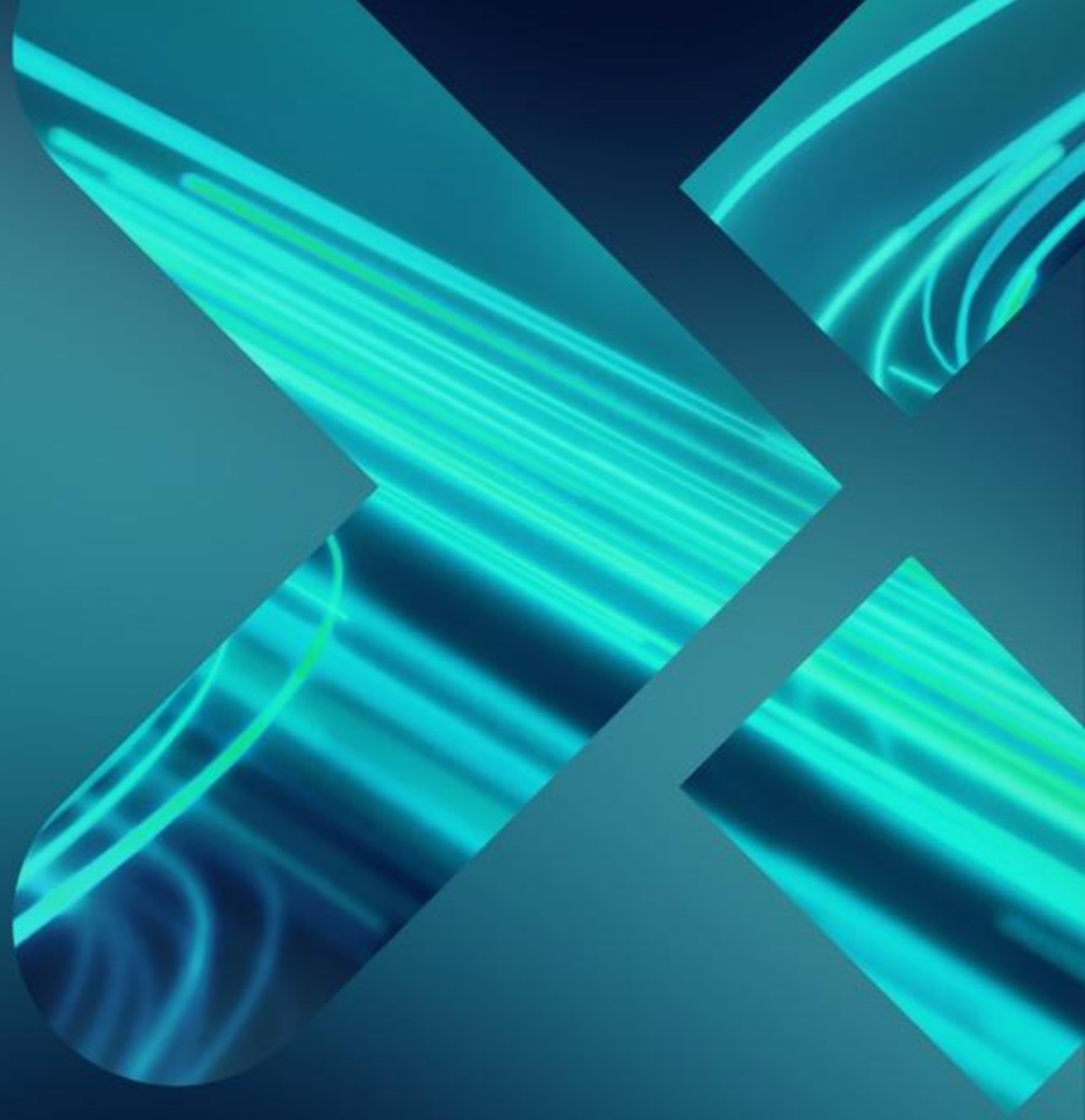
When we write custom clang-tidy checks

How we write idiomatic custom checks:

- A useful workflow
- Idiomatic use of built-in matchers
- Writing idiomatic custom matchers

Applying fixes at scale:

- Fixits, less useful than you’d think?
- Keeping track of NOLINTs

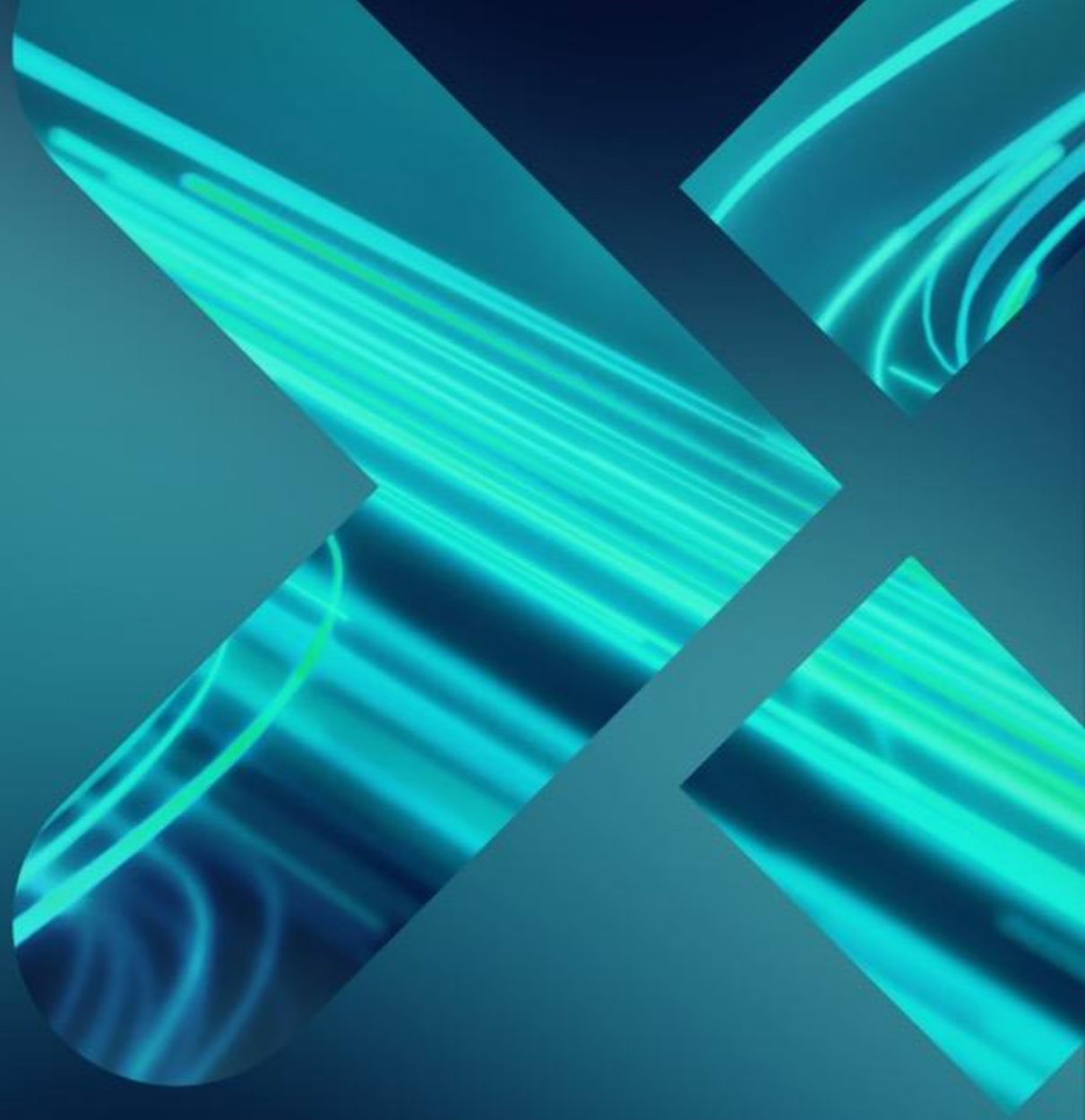


SIEMENS

When we write custom clang-tidy checks

How we write idiomatic custom checks

Applying fixes at scale



SIEMENS

Why We Write Custom clang-tidy Checks

Custom clang-tidy checks:

- Integrate well with our LLVM-based workflow
- Are as powerful as LLVM
- Are surprisingly easy to write:
  - Can be opinionated
  - Don’t have to be fast
  - Don’t have to give 100% coverage
  - Can give occasional false positives

As of 2026-02-27 we have 27 custom checks

SIEMENS

Use the right tool

Write a custom check if…

1. We have a simple, local rule to enforce
2. We need compiler-level information
3. We expect the problem to recur

Nice-to-haves:

- 100% coverage
- No false positives
- Automatic fixits

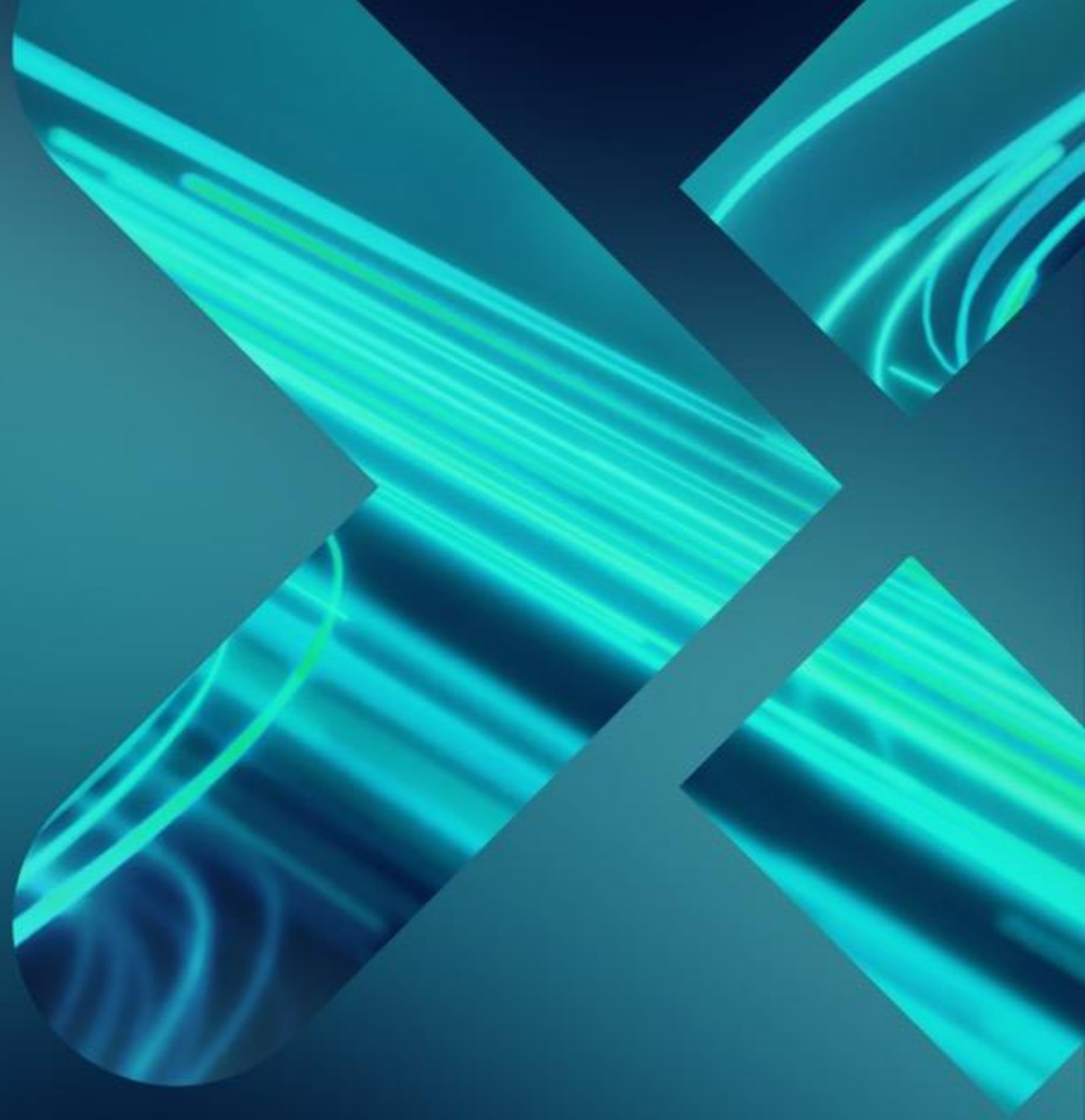
Use other tools if…

1. The rule can be enforced at the C++ level
   - Always the best solution
   - Use libtooling for large-scale refactoring
2. We require cross-TU analysis
   - Look at clang static analyser
   - …unless you can write a simple, local rule and call it “bugprone”

When we write custom clang-tidy checks

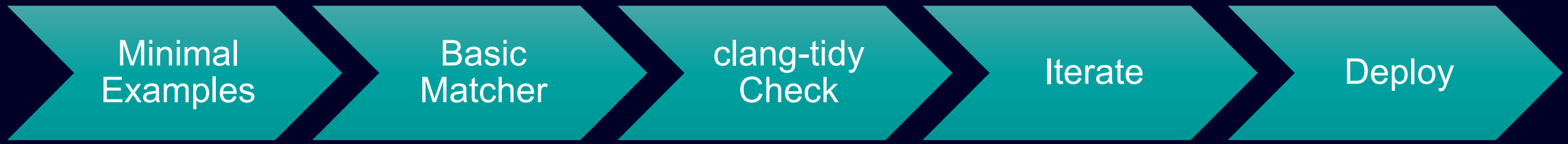
How we write idiomatic custom checks

Applying fixes at scale



SIEMENS

Test-Driven Development (TDD) for Custom clang-tidy Checks



- As complete as possible
- Include patterns and antipatterns

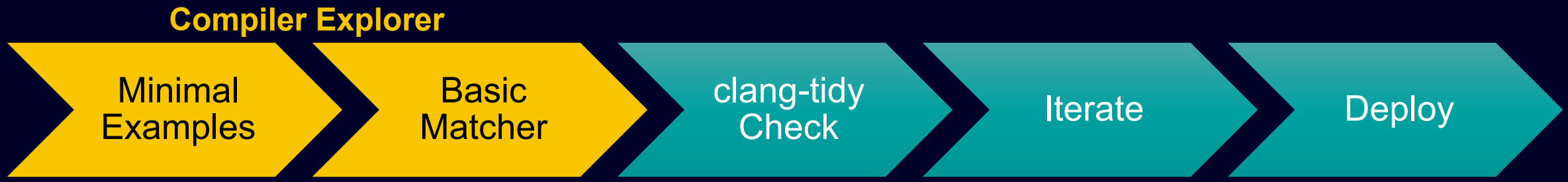
- Just trying to get “close enough”

- Minimal examples become unit tests
- Start from the basic matcher
- Write the check function

- If your unit tests are good enough, this should just work…

SIEMENS

Test-Driven Development (TDD) for Custom clang-tidy Checks



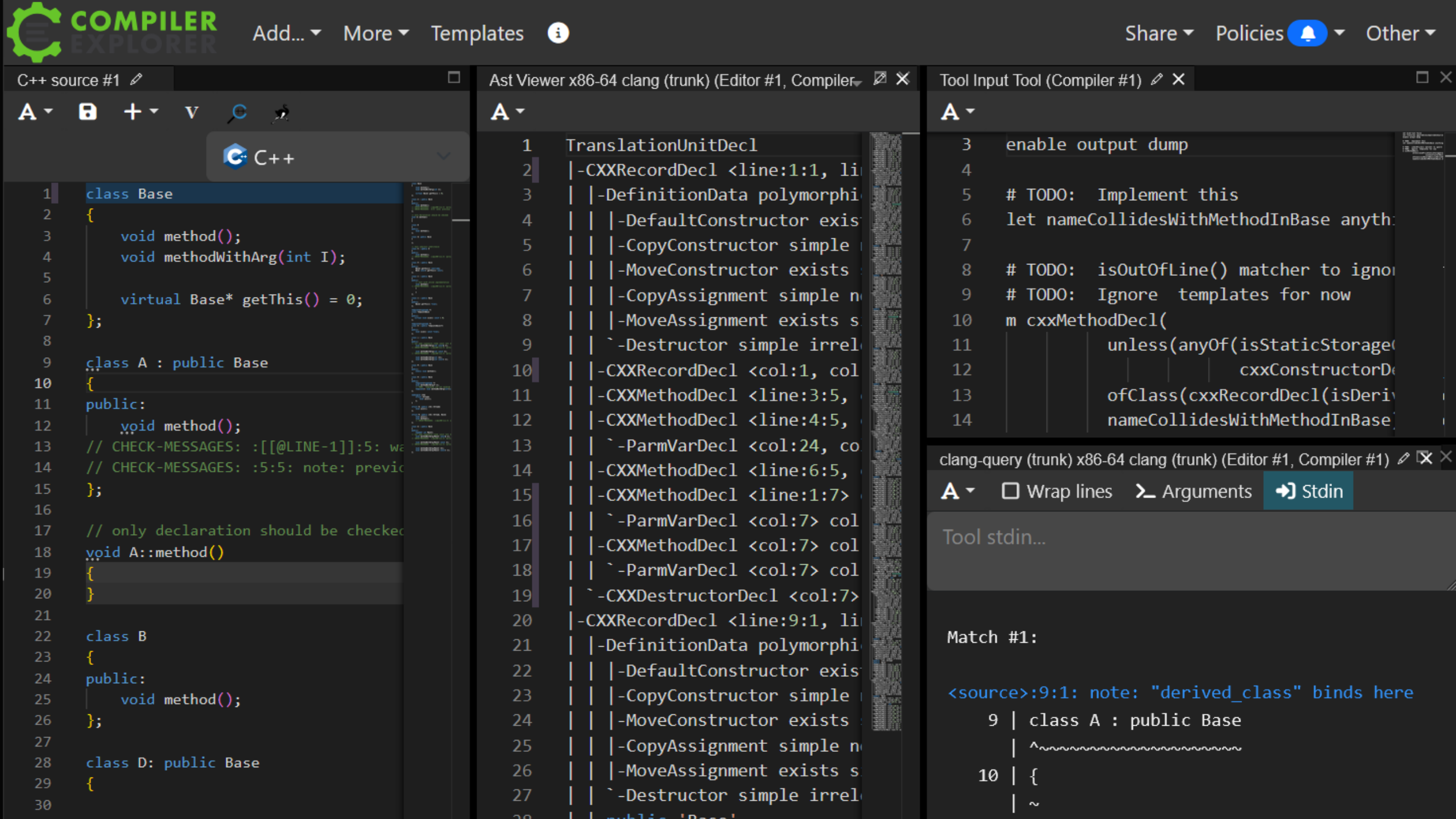
- As complete as possible
- Include patterns and antipatterns

- Just trying to get “close enough”

- Minimal examples become unit tests
- Start from the basic matcher
- Write the check function

- If your unit tests are good enough, this should just work…

SIEMENS


COMPILER EXPLORER
Add...
More
Templates
Share
Policies
Other
C++ source #1
C++
class Base
{
void method();
void methodWithArg(int I);
virtual Base* getThis() = 0;
};
class A : public Base
{
public:
void method();
// CHECK-MESSAGES: :[[@LINE-1]]:5: wa
// CHECK-MESSAGES: :5:5: note: previc
};
// only declaration should be checkec
void A::method()
{
}
class B
{
public:
void method();
};
class D: public Base
{
Ast Viewer x86-64 clang (trunk) (Editor #1, Compiler
TranslationUnitDecl
|-CXXRecordDecl <line:1:1, li
| |-DefinitionData polymorphi
| | |-DefaultConstructor exis
| | |-CopyConstructor simple
| | |-MoveConstructor exists
| | |-CopyAssignment simple n
| | |-MoveAssignment exists s
| | `-Destructor simple irrel
| |-CXXRecordDecl <col:1, col
| |-CXXMethodDecl <line:3:5,
| |-CXXMethodDecl <line:4:5,
| | `-ParmVarDecl <col:24, co
| |-CXXMethodDecl <line:6:5,
| |-CXXMethodDecl <line:1:7>
| | `-ParmVarDecl <col:7> col
| |-CXXMethodDecl <col:7> col
| | `-ParmVarDecl <col:7> col
| `-CXXDestructorDecl <col:7>
|-CXXRecordDecl <line:9:1, li
| |-DefinitionData polymorphi
| | |-DefaultConstructor exis
| | |-CopyConstructor simple
| | |-MoveConstructor exists
| | |-CopyAssignment simple n
| | |-MoveAssignment exists s
| | `-Destructor simple irrel
Tool Input Tool (Compiler #1)
enable output dump
TODO: Implement this
let nameCollidesWithMethodInBase anyth
TODO: isOutOfLine() matcher to igno
TODO: Ignore templates for now
m cxxMethodDecl(
unless(anyOf(isStaticStorage
cxxConstructorD
ofClass(cxxRecordDecl(isDeri
nameCollidesWithMethodInBase
clang-query (trunk) x86-64 clang (trunk) (Editor #1, Compiler #1)
Wrap lines
Arguments
Stdin
Tool stdin...
Match #1:
<source>:9:1: note: "derived_class" binds here
9 | class A : public Base
| ^~~~~~~~~~~~~~~~~~~~~~
10 | {
| ~

Idiomatic Use of Built-In Matchers

Fail fast

- Match least common nodes first
- Put expensive matchers last
  - Narrowing matchers tend to be cheaper than traversal matchers
  - `*each*` matchers tend to be eager
- Filter in the matcher rather than the check
- Prefer anyOf instead of multiple matchers

Avoid going up the AST

- The AST is a downwards tree
- Accessing the upwards tree can be expensive
  - Avoid hasParent(), hasAncestor()
  - Prefer hasDeclContext(), ofClass(), etc.

Strive for clarity

- Write really good mock-ups
- Pick the right traversal kind
- Careful when matching instantiated vs uninstantiated templates
- Binding inside anyOf() will only bind to the first instance
- Be clear about when to write custom matchers

When do we write our own matchers?

Built-in matchers

Documented matchers are stable, but the AST isn't

- Not everything is exposed in the matcher API
- Some things are not possible at the matcher API level
- Fill in the gaps!

Custom matchers

1. Access to LLVM internals, class-level information
2. Very few restrictions: Matchers are just callables

Ways to Spell Matchers: Built-in Node Matchers and Aliases

Built-in node matchers:

```
const internal::VariadicDynCastAllOfMatcher<Decl,
PragmaCommentDecl>
   pragmaCommentDecl;
```

Aliases:

Clang-query: let nameCollidesWithMethodInBase hasName("method")

C++: `auto const nameCollidesWithMethodInBase = hasName("method");`

Ways to Spell Matchers : Macros from ASTMatchersMacros.h

```
// These matchers are from clang/include/clang/ASTMatchers/ASTMatchers.h
// See clang/include/clang/ASTMatchers/ASTMatchersMacros.h for macro definitions


// cxxMethodDecl(isConst())
AST_MATCHER(CXXMethodDecl, isConst) {
  return Node.isConst();
}


//cxxMethodDecl(ofClass(hasName("ParentClass")))
AST_MATCHER_P(CXXMethodDecl, ofClass,
              internal::Matcher<CXXRecordDecl>, InnerMatcher) {

  ASTChildrenNotSpelledInSourceScope RAII(Finder, false);

  const CXXRecordDecl *Parent = Node.getParent();
  return (Parent != nullptr && InnerMatcher.matches(*Parent, Finder, Builder));
}
```

Writing Idiomatic Custom Matchers

They’re just matchers

- Be lazy
- Avoid going up the AST
- Strive for clarity between the matcher API level and the LLVM class level

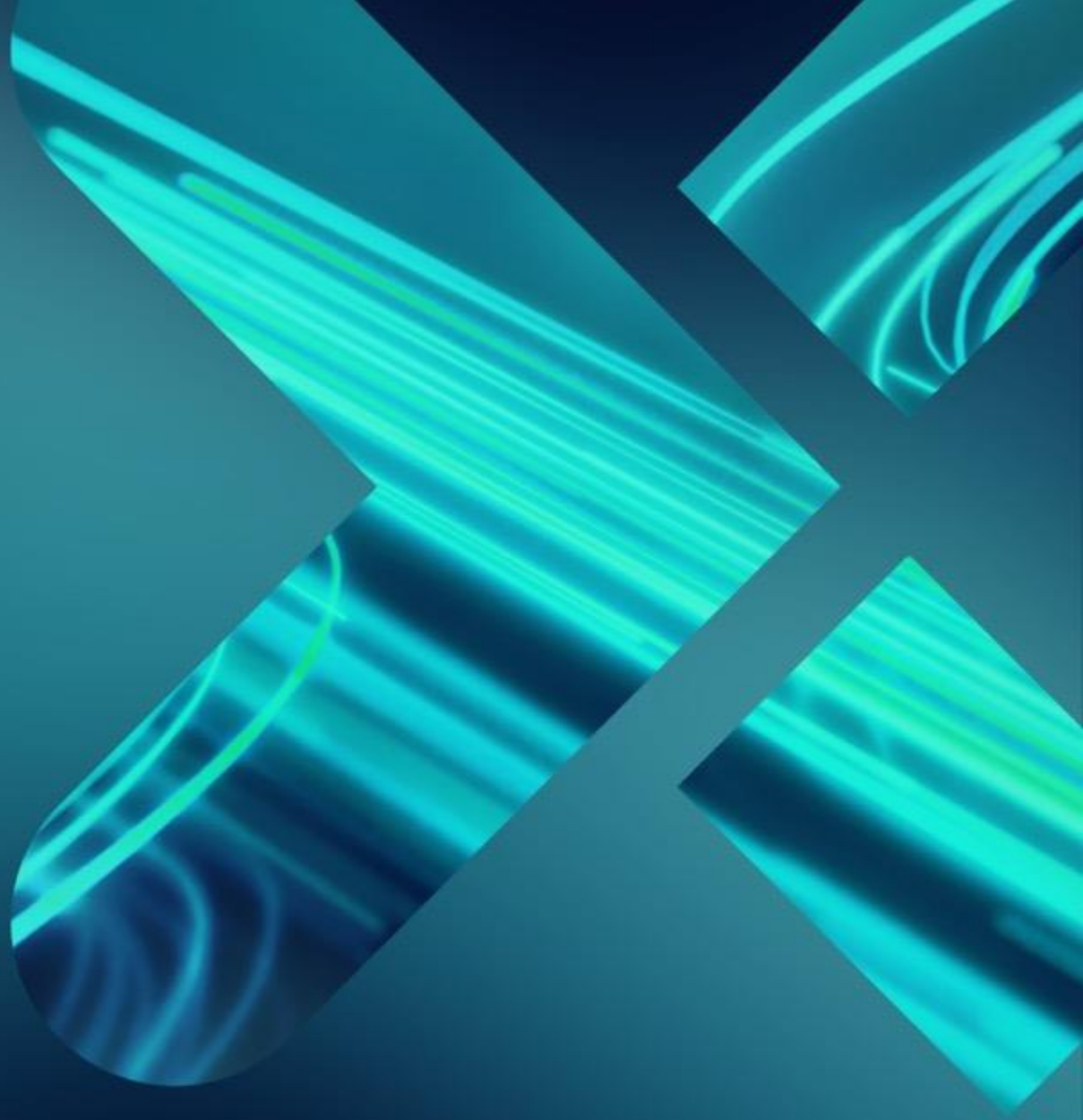
Get good advice

- Read the LLVM docs
- Compare with built-in matchers
- Don’t miss clang-tidy/utils
- Ask on discord
- Upstream where possible

When we write custom clang-tidy checks

How we write idiomatic custom checks

Applying fixes at scale



SIEMENS

Strategies for implementing new checks

Fix everything immediately

- The best solution
- Tricky in a fast-moving codebase that requires many fixes
- Automatic fixits help!

NOLINT everything, fix later

- Stops new warnings being introduced
- It's easy to automatically add NOLINTs
- Consider democratising fixes
- Requires NOLINT tracking and metadata

Start lenient check, get stricter

- Stops new warnings being introduced
- Situationally useful
  - Consider ignoring templates, macros

Directory-by-directory rollout

- Untested in Simcenter STAR-CCM+
- Different coding standards in different parts of the codebase
- Multiple .clang-tidy files add complexity
- Headers can be included across directories

FixIts: Less useful than you'd think?

For

- Useful when applying many fixes
- Convenient in IDEs
- Only need to be syntactically correct
- Can be partial:
  - Fix only a subset
  - Generate incomplete code
  - Can be emitted as notes

Against

- Large numbers of fixes are only applied once per codebase
- Writing fixits can be hard or impossible
  - Only ~50% of upstream checks have fixits
- Value scales with number of hits
- Plausible-looking but incorrect fixits are dangerous

Strategies for implementing new checks

Fix everything immediately

- The best solution
- Tricky in a fast-moving codebase that requires many fixes
- Automatic fixits help!

NOLINT everything, fix later

- Stops new warnings being introduced
- Consider democratising fixes
- It's easy to automatically add NOLINTs
- Requires NOLINT tracking and metadata

Start lenient check, get stricter

- Stops new warnings being introduced
- Situationally useful
  - Consider ignoring templates, macros

Directory-by-directory rollout

- Untested in Simcenter STAR-CCM+
- Different coding standards in different parts of the codebase
- Multiple .clang-tidy files add complexity
- Headers can be included across directories

Keeping Track of NOLINTs

```
// NOLINT(some-check)
```

- Is this a false positive?
- Should this be fixed?
- Is a fix scheduled?
- Accompanying comments are not searchable

Keeping Track of NOLINTs

```
// NOLINT(some-check)
```

- Is this a false positive?
- Should this be fixed?
- Is a fix scheduled?
- Accompanying comments are not searchable

```
// NOLINT(some-check) [WONT_FIX]
// NOLINT(some-check) [ticket-1234]
```

- Easily enforced with text-based tools
- Can include any arbitrary metadata
- Compatible with upstream NOLINT statements
- Ticket numbers can be sanity checked with ticketing system APIs
- When enforced, all NOLINTs become traceable

Currently no proposal to upstream this

Strategies for implementing new checks

Fix everything immediately

- The best solution
- Tricky in a fast-moving codebase that requires many fixes
- Automatic fixits help!

NOLINT everything, fix later

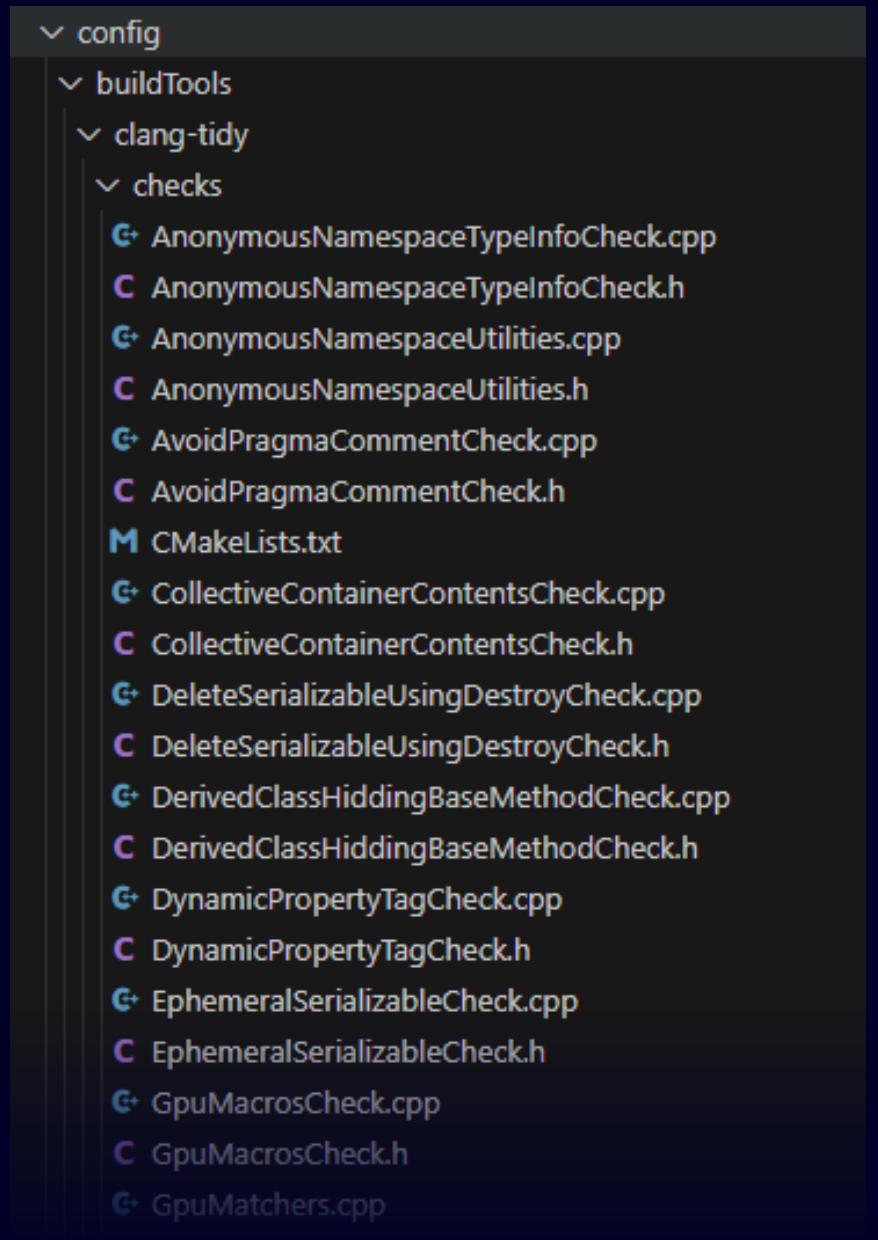
- Stops new warnings being introduced
- It's easy to automatically add NOLINTs
- Consider democratising fixes
- Requires NOLINT tracking and metadata

Start lenient check, get stricter

- Stops new warnings being introduced
- Situationally useful
  - Consider ignoring templates, macros

Directory-by-directory rollout

- Untested in Simcenter STAR-CCM+
- Different coding standards in different parts of the codebase
- Multiple .clang-tidy files add complexity
- Headers can be included across directories


config
buildTools
clang-tidy
checks
AnonymousNamespaceTypeInfoCheck.cpp
AnonymousNamespaceTypeInfoCheck.h
AnonymousNamespaceUtilities.cpp
AnonymousNamespaceUtilities.h
AvoidPragmaCommentCheck.cpp
AvoidPragmaCommentCheck.h
CMakeLists.txt
CollectiveContainerContentsCheck.cpp
CollectiveContainerContentsCheck.h
DeleteSerializableUsingDestroyCheck.cpp
DeleteSerializableUsingDestroyCheck.h
DerivedClassHiddingBaseMethodCheck.cpp
DerivedClassHiddingBaseMethodCheck.h
DynamicPropertyTagCheck.cpp
DynamicPropertyTagCheck.h
EphemeralSerializableCheck.cpp
EphemeralSerializableCheck.h
GpuMacrosCheck.cpp
GpuMacrosCheck.h
GpuMatchers.cpp

SIEMENS

Contact

Tom James
tom.james@siemens.com

SIEMENS