Compiler as a Service: C++ Goes Live

Interactive C++, language interoperability and beyond

Aaron Jomy, Vipul Cariappa
for the Compiler Research group, ROOT Project - CERN

`using std::cpp`
Universidad Carlos III de Madrid, 03/17/2026

EP-SFT
Software Frameworks and Tools

PRINCETON
UNIVERSITY

Contents

- Introduction
- Compiler as a Service (CaaS)
  - Insights on Incremental Compilation
  - Building a CaaS with examples
- CppInterOp
- Applications and Demo

Motivation

Can we combine power of C++ with the expressiveness of Python?

Motivation

Exploratory programming

- Converge from prototype to production

Rapid Iteration

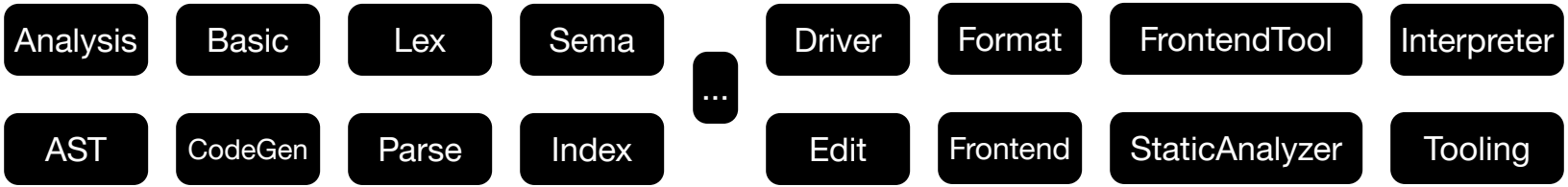
- Evolution of classes/data structures without recompilation
- Redefining entities

Interactivity

- Interactive access of your data
- Interactive usage of a C++ library -> Figure out what works best
- Learning new frameworks

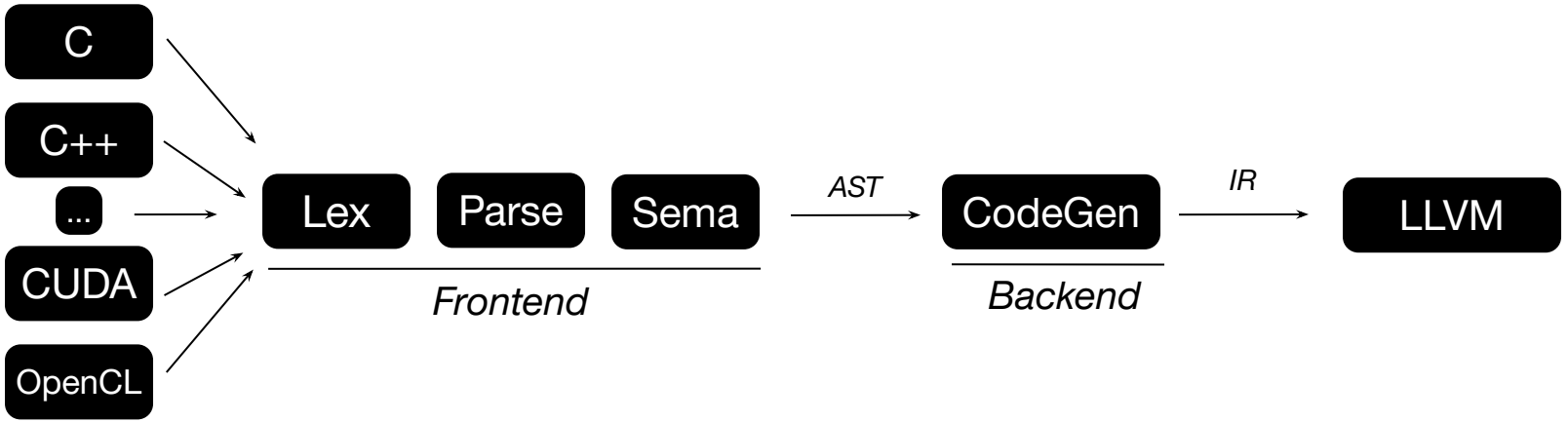
The Clang Project

- Clang is a compiler which supports C, C++, Objective-C, and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript, CUDA, SYCL, and HIP frameworks.
- Just like LLVM, Clang is built by a set of reusable components and can be used as a library.



The Clang Project

Clang takes input pipes it through a frontend and a backend and produces machine code



Compiler-As-A-Service

How to use Clang for incremental compilation

Using Clang As A Library

AST

Tooling

ex1.cpp: Extracting documentation from source

```
#include "clang/AST/Comment.h"
#include "clang/AST/DeclTemplate.h"
#include "clang/Tooling/Tooling.h"
...
auto ASTU = tooling::buildASTFromCodeWithArgs(Code,
{“-std=c++20”});
auto &C = ASTU->getASTContext();
auto *TU = C.getTranslationUnitDecl();
TU->dump();
```

Run the compiler on given code.

```
|-ClassTemplateDecl 0x7f83db895f48 <input.cc:4:1, line:5:40> col:8 ComplexNumber
| |-TemplateTypeParmDecl 0x7f83db895dd0 <line:4:10, col:19> col:19
| |-CXXRecordDecl 0x7f83db895e98 <line:5:1, col:40> col:8 struct ComplexNumber
| | ...
| | |-FullComment 0x7f83dc00c200 <line:2:4, col:52>
| | | `-ParagraphComment 0x7f83dc00c1d0 <col:4, col:52>
| | |   `-TextComment 0x7f83dc00c1a0 <col:4, col:52>
        Text=" This is the documentation for the ComplexNumber."
```

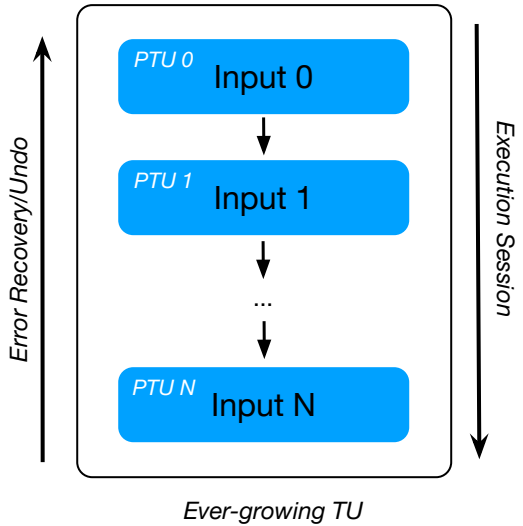
A Simple C++ REPL

Frontend

Interpreter

Incremental Compilation in Clang

- We can split the translation unit into a sequence of partial translation units (PTU)
- Processing a PTU might extend an earlier PTU (template instantiation)



Incremental Compilation in Clang

```
// Initialize our builder class.
clang::IncrementalCompilerBuilder CB;

CB.SetCompilerArgs({"-std=c++20"}); // pass `-xc` for C.

// Create the incremental compiler instance.
auto CI = ExitOnErr(CB.CreateCpp());

// Create the interpreter instance.
auto Interp = ExitOnErr(Interpreter::create(std::move(CI)));
```

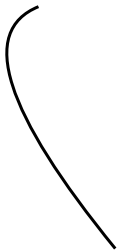
-fplugin=my_plugin.so

Standard compiler flags

Creates an incremental compiler instance

Incremental Compilation in Clang

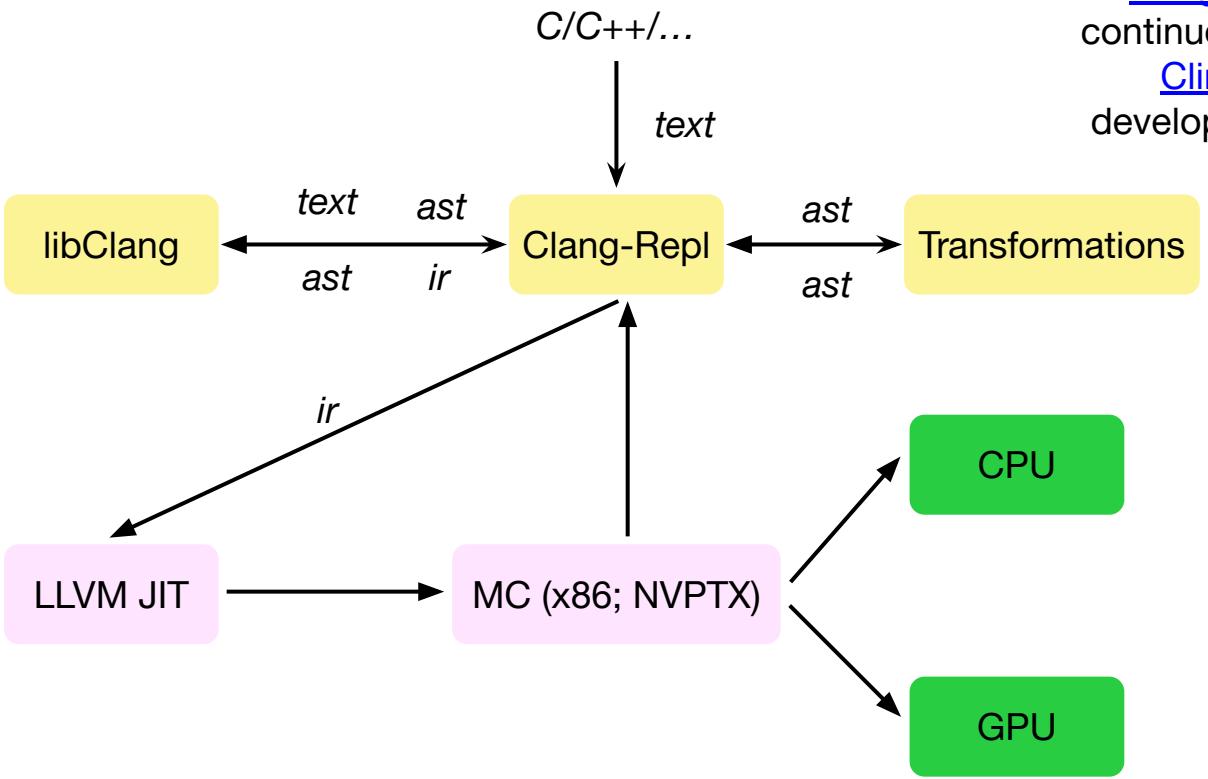
```
llvm::LineEditor LE("using-std-cpp-repl");
while (std::optional<std::string> Line = LE.readLine()) {
  if (*Line == "%quit")
    break;
  if (auto Err = Interp->ParseAndExecute(*Line)) {
    …
  }
}
```



Adds a partial translation unit

REPL - Data Flow

clang-repl is the result of continuous upstreaming of the Cling C++ interpreter developed at CERN, to LLVM



Bridging Compiled and Interpreted C++

Frontend

Interpreter

ex3.cpp: Bridging Compiled and Interpreted C++

An efficient, ref-counted, small-buffer optimized facility to transport results
Supports pretty printing and type conversion operations

```
// Create a value to store the transport the result from JIT.
clang::Value V;
Interp->ParseAndExecute(R"(extern "C" int sq(int x){return x*x;}
                           sq(12))", &V);

printf("From JIT: square(12)=%d\n", V.getInt());


// Or just get a function pointer and call it in compiled code:
auto SymAddr = ExitOnErr(Interp->getSymbolAddress("sq"));
auto sqPtr = SymAddr.toPtr<int(*)(int)>();
printf("From compiled code: square(13)=%d\n", sqPtr(13));
```

Programmatically Instantiating C++ Templates

- Create a library containing CaaS building blocks and create an interface for it
- Build a C binary that can programmatically instantiate and call a C++ template function

```
#include <typeinfo>
void* operator new(__SIZE_TYPE__, void* __p) noexcept;

extern "C" int printf(const char*, ...);

class A {};
struct B {
    template <typename T>
    static void callme(T) {
        printf(" Instantiated with [%s] \n", typeid(T).name());
    }
};
```

ex4-lib: A Library Capable of Instantiating a Template

```
void Clang_Parse(const char* Code);

void* Clang_LookupName(...);
unsigned Clang_GetFunctionAddress(...);
void* Clang_CreateObject(...);
void* Clang_InstantiateTemplate(...);
```

Returns a declaration given a string

Returns the in-memory address of JIT'd function

Allocates storage for a declaration

ex4-lib: A Library Capable of Instantiating a Template

```
int main(int argc, char **argv) {
  Clang_Parse(Code);

  Decl_t TemplatedClass = Clang_LookupName("B", /*Context=*/0);

  // Instantiate B::callme with the given types
  Decl_t Instantiation = Clang_InstantiateTemplate(TemplatedClass, "callme", "A");

  // Get the symbol to call
  typedef void (*fn_def)(void*);
  fn_def callme_fn_ptr = (fn_def) Clang_GetFunctionAddress(Instantiation);

  // Create objects of type A
  Decl_t T = Clang_LookupName("A", /*Context=*/0);
  void* NewA = Clang_CreateObject(T);

  callme_fn_ptr(NewA);
}
```

Bridging interpreted C++ with other environments

Teaching Python C++

Let’s build a Python wrapper API for the the functions in ex4-lib:

- InterOpLayerWrapper — responsible for C++ template instantiations
- TemplateWrapper — Finds and matches C++ template arguments
- CallCPPFunc — Invokes the C++ template function

Teach Python to express standard C++: std.vector[“int”]

Teaching Python C++: Instantiating Templates on-demand

```python
import ctypes

libInterop = ctypes.CDLL("ex4-lib.so")
# tell ctypes which function to call and what are the expected
param/return types.
_cpp_compile = libInterop.Clang_Parse
_cpp_compile.argtypes = [ctypes.c_char_p]

def cpp_compile(arg):
  return _cpp_compile(arg.encode("ascii"))
# define some classes to play with
cpp_compile(r"""\
class A {};
struct B : public A {
  template<typename T> static void callme(T*) {
   printf("Template: %s\n", typeid(T).name());
 }
};
""")
```

```
void Clang_Parse(const char*
Code);
```

Teaching Python C++: Instantiating Templates on-demand

```
class InterOpLayerWrapper:
  # Provide a python wrapper over the libInterop layer.
  _get_scope = libInterop.Clang_LookupName
  _get_scope.restype = ctypes.c_void_p
  _get_scope.argtypes = [ctypes.c_char_p]
  _construct = libInterop.Clang_CreateObject
```

```
class TemplateWrapper:
  # Find, instantiate and invoke a template function.
  def __init__(self, scope, name):
    …
  def __getitem__(self, *args, **kwds):
    # Look up the template and return the overload.
    return gIL.get_template(
      self._scope, self._name, tmpl_args = args)
```

```
class CallCPPFunc:
  # Responsible for calling low-level function pointers.
  _get_funcptr = libInterop.Clang_GetFunctionAddress
  _get_funcptr.restype = ctypes.c_void_p
```

```
void* Clang_LookupName(...);
void* Clang_CreateObject(...);
void* Clang_InstantiateTemplate(...);
unsigned Clang_GetFunctionAddress(...);
```

Overload access to array-like entries using square brackets

`my_list[2]` internally calls `my_list.__getitem__(2)` to return the value

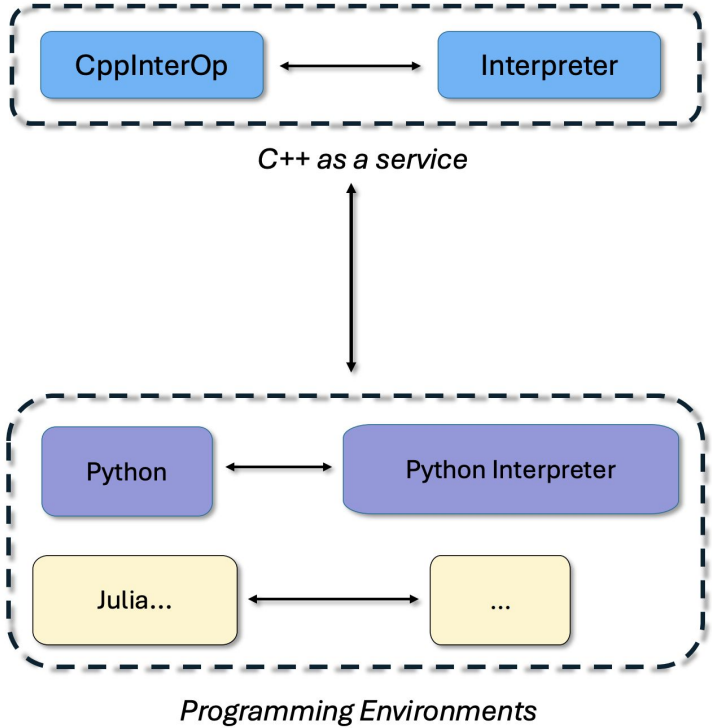
Teaching Python C++: Python Metaclasses

```
gIL = InterOpLayerWrapper()# Initialize our C++ interoperability layer wrapper

# Create a couple of types to play with
CppA = type('A', (), {
  'handle'   : gIL.get_scope('A'),
  '__new__'  : cpp_allocate
})
h = gIL.get_scope('B')
CppB = type('B', (A,), {
   'handle'   : h,
   '__new__'  : cpp_allocate,
   'callme'   : TemplateWrapper(h, 'callme')
})
# Connect to C++ classes
a = CppA()
b = CppB()
# Explicit template instantiation and execution
b.callme['A'](a)
# Implicit template instantiation and execution
b.callme(b)
```

CppInterOp

- Lightweight layer on top of LLVM/Clang
- Provides efficient, on-demand reflection and incremental JIT compilation capabilities
- Embeds Clang and LLVM as libraries in your framework, in a backward compatible way
- Supports downstream tools with reflection API for interactive C++ and language interoperability



https://cppinterop.readthedocs.io/
https://github.com/compiler-research/CppInterOp

CppInterOp API

```
Cpp::Declare(R"(
        template<typename T>
        struct S {
          bool operator<(T &a) { return 0 < a; }
        };
    )", Decls);
ASTContext& C = Interp->getCI()->getASTContext();
auto Instance1 = Cpp::InstantiateTemplate(Decls[0],
                 {C.IntTy.getAsOpaquePtr()}, 1);
auto* obj = Cpp::Construct(Instance1);
std::vector<TCppFunction_t> ops;
Cpp::GetOperator(Instance1, Cpp::Operator::OP_Less, ops);
Cpp::JitCall FCI_Add = Cpp::MakeFunctionCallable(ops[0]);
int a = 5;
bool result = false;
void* args[1] = {(void*)&a};
FCI_Add.Invoke(&result, {args, /*arg_count=*/1}, obj);
assert(result == true);
```

```
Cpp::LoadLibrary("mylib.so");
Cpp::Declare("#include \"mylib.h\"");
auto f = reinterpret_cast<int (*)(int,
int)>(Cpp::GetFunctionPointer("add"));
assert(f(2, 3) == 5);
```

```
extern "C" void __jc_1(void* obj, unsigned
long nargs, void** args, void* ret)
{
   if (ret) {
      new (ret) (bool)
(((S<int>*)obj)->operator<((int&)*(int*)arg
s[0]));
      return;
   }
   else {
//..
```

CppInterOp Enables Device Acceleration

CppInterOp is based on on Clang and the LLVM JIT

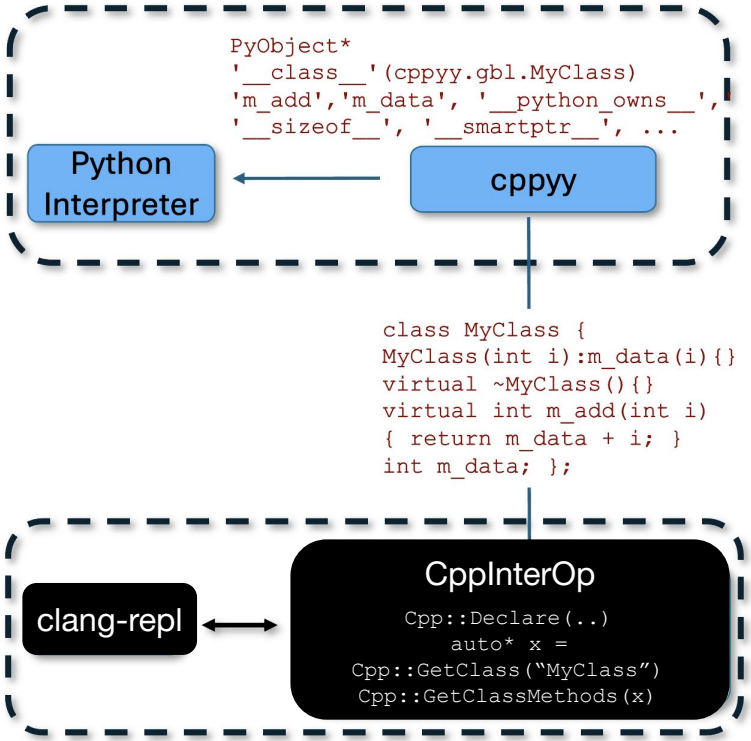
Supports hardware accelerators or parallel programming API:

- `Cpp::CreateInterpreter("--std=c++20", "--cuda")`
- `Cpp::CreateInterpreter("--std=c++17", "-fopenmp")`

Downstream Tools

cppyy

```
>>> import cppyy
>>> cppyy.cppdef("""
... class MyClass {
... public:
...     MyClass(int i) : m_data(i) {}
...     virtual ~MyClass() {}
...     virtual int add_int(int i) { return m_data + i; }
...     int m_data;
... };""")
True
>>> from cppyy.gbl import MyClass
>>> m = MyClass(42)
>>> cppyy.cppdef("""
... void say_hello(MyClass* m) {
...     std::cout << "Hello, the number is: " << m->m_data << std::endl;
... }""")
True
>>> MyClass.say_hello = cppyy.gbl.say_hello
>>> m.say_hello()
Hello, the number is: 42
>>> m.m_data = 13
>>> m.say_hello()
Hello, the number is: 13
>>> class PyMyClass(MyClass):
...     def add_int(self, i):  # python side override (CPython only)
...         return self.m_data + 2*i
...
>>> cppyy.cppdef("int callback(MyClass* m, int i) { return m->add_int(i); }")
True
>>> cppyy.gbl.callback(m, 2)             # calls C++ add_int
15
>>> cppyy.gbl.callback(PyMyClass(1), 2)  # calls Python-side override
5
>>>
```



The original cppyy project (github.com/wlav/cpyyy) is based on Cling
Compiler-Research moved cppyy to CppInterOp and clang-repl
Releases on pip coming soon: https://github.com/compiler-research/CppJIT

jank-lang

```
(ns jank.json
  (:include "fstream"
            "./json.hpp")
  (:refer-global :only [std.ifstream nlohmann.json.parse]
                 :rename {std.ifstream open-file
                          nlohmann.json.parse parse-json}))

(defn -main [& args]
  (let [file (open-file (cpp/cast std.string (first args)))
        json (parse-json file)]
    (println (.dump json 2))))
```

Dialect of Clojure built on LLVM

Extends CppInterOp for C++ Interoperability (AOT, extra predicates and transformations)

Allows seamlessly writing C++ and Clojure in the same file.

First-class support for C++ includes, bringing C++ globals into a jank namespace and renaming them.

https://jank-lang.org/

https://github.com/jank-lang/jank

CppInterOp.jl

```
julia> import CppInterOp as Cpp

julia> using Test

julia> I = Cpp.create_interpreter()
CppInterOp.Interpreter(Ptr{CppInterOp.LibCppInterOp.CXInterpreterImpl}(0x000000001c2c0c60))

julia> Cpp.process(I, "int x = 1 + 1;") # PTU_1 created
true

julia> Cpp.evaluate(I, "x")             # PTU_2 created by evaluate
2

julia> Cpp.undo(I, 2)
true

julia> Cpp.process(I, "float x = 42.0;")
true

julia> Cpp.evaluate(I, "x")
42.0f0
```

Thanks to Yupei Qi (Gnimuc) for his contributions!

Recipe: *> pkg add https://github.com/Gnimuc/CppInterOp.jl*

xeus-cpp

Jupyter kernel for C++ based on the native implementation of the Jupyter protocol xeus

Successor to the xeus-cling project

Leverages CppInterOp for clang-repl and CaaS facilities

xeus-cpp-lite extends this model to the browser with WebAssembly and JupyterLite

Thanks to the QuantStack team!

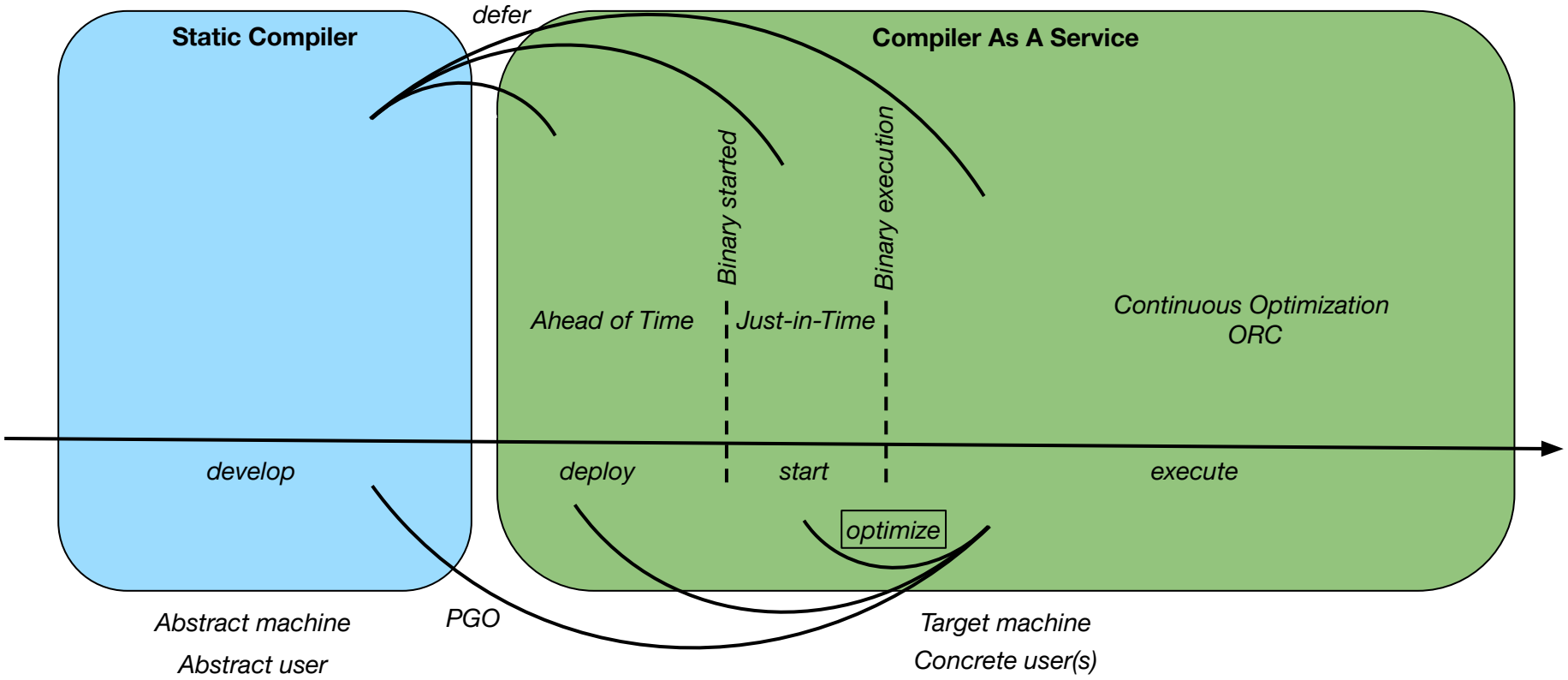
https://github.com/compiler-research/xeus-cpp

Demo

- Interactive C++ notebooks with xeus-cpp
  - sub-interpreters with different C++ standards, OpenMP
  - Running CUDA in notebooks: https://compiler-research.org/CppInterOp/lab/index.html
- C++ execution in the browser with WebAssembly and xeus-cpp-lite
- Putting it all together - Interactive Exploratory Programming Workflow

Putting it together: Interactive exploratory programming demo

Summary: CaaS Interaction And Opportunities



About us

Our compiler-research.org initiatives aim to:

- Make compiler research visible and connected
- Train and mentor the next generation of compiler engineers
- Support impactful open-source R&D
- Build bridges between academia, industry, and scientific domains

Internship Opportunities 2026 @ CR

@

We are happy to announce our Clang-focused fellowship program sponsored by CppAlliance:

- Remote, GSoC-like, project-based
- Upto 6-month internships on Clang oriented projects
- https://compiler-research.org/open_projects?tag=cppalliance-fellow-26
- Application form: https://forms.gle/hTCqC2nkTZwqNabc6

Thank You!

Slides and WebAssembly demo are hosted at https://aaronj0.github.io/compiler-as-a-service-talk/

vgvassilev@cern.ch

aaron.jomy@cern.ch

vipul.cariappa.thimmaiah@cern.ch

Programming model

Prototype in Python, compute in C++ and CUDA – all in one runtime.