Advancing Python-C++ Interoperability in ROOT *and beyond*

Aaron Jomy[1], *Vipul Cariappa*[1, 2]
for the ROOT Project

[1]*CERN EP-SFT,* [2]*Ramaiah University of Applied Sciences*
13th ROOT Users Workshop
18-11-2025

EP-SFT
Software Frameworks and Tools

Introduction: ROOT & Python

ROOT's Python interface allows the creation of bindings between Python and C++ in a ***dynamic and automatic way***.

How?

An automatic Python-C++ bindings engine which powers ROOT's Python interoperability

The standalone Cppyy package can be pip installed

cppyy.readthedocs.io

tutorials/dataframe/
df002_dataModel.py

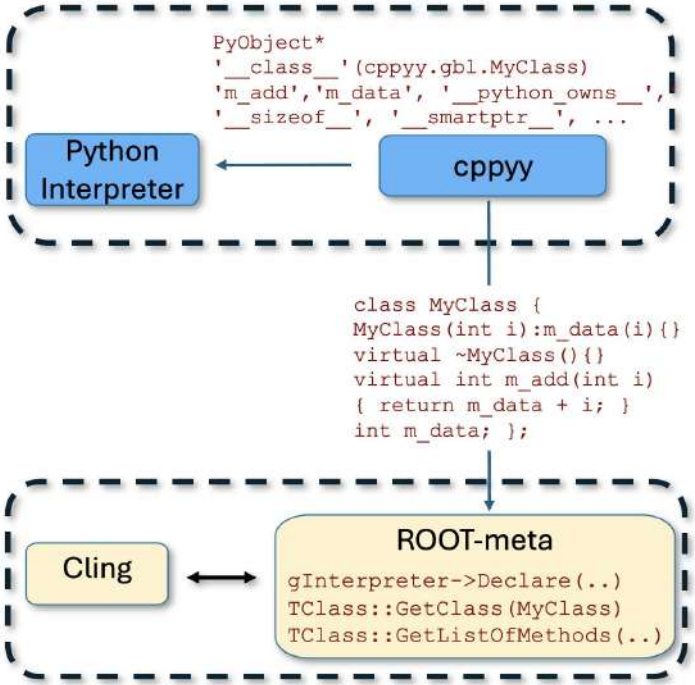
```
[ ]: import ROOT

[ ]: # A simple helper function to fill a test tree
fill_tree_code = """
using FourVector = ROOT::Math::XYZTVector;
using FourVectorVec = std::vector<FourVector>;
using CylFourVector = ROOT::Math::RhoEtaPhiVector;

void fill_tree(const char *filename, const char *treeName) {
   const double M = 0.13957; // set pi+ mass
   TRandom3 R(1);

   auto genTracks = [&](){
      FourVectorVec tracks;
      const auto nPart = R.Poisson(15);
      tracks.reserve(nPart);
      for (int j = 0; j < nPart; ++j) {
         const auto px = R.Gaus(0, 10);
         const auto py = R.Gaus(0, 10);
         const auto pt = sqrt(px * px + py * py);
         const auto eta = R.Uniform(-3, 3);
         const auto phi = R.Uniform(0.0, 2 * TMath::Pi());
         CylFourVector vcyl(pt, eta, phi);
         // set energy
         auto E = sqrt(vcyl.R() * vcyl.R() + M * M);
         // fill track vector
         tracks.emplace_back(vcyl.X(), vcyl.Y(), vcyl.Z(), E);
      }
      return tracks;
   };

   ROOT::RDataFrame d(64);
   d.Define("tracks", genTracks).Snapshot(treeName, filename, {"tracks"});
}
"""

[ ]: # Prepare an input tree to run on
fileName = "df002_dataModel_py.root"
treeName = "myTree"
ROOT.gInterpreter.Declare(fill_tree_code)
ROOT.fill_tree(fileName, treeName)

# We read the tree from the file and create a RDataFrame
d = ROOT.RDataFrame(treeName, fileName)
```

cppyy

Leverages Cling as a runtime interpreter, enabling the interactive execution of C++

The standalone Cppyy package (upstream) relies on ROOT for its interpreter and reflection layer

ROOT uses a customized fork of Cppyy, to support the HEP use case



Compiler level interactions with the Type System (ROOT-meta)

- ROOT-meta: an essential layer enabling the powerful ROOT I/O, and other ROOT components. Uses LLVM and Clang to drive its C++ reflection system.
- The same system is utilized by Cppyy and ROOT's Python bindings
- Leads to certain limitations in *template instantiation*, *overload resolution*, *enum support*
- Uses more memory than required

Divergence and maintenance costs

- The cross-dependency between ROOT and Cppyy makes it harder to adopt upstream features that rely on patches to the type system
  - For example, changes that don't work well with the I/O
- Complicates the development of new features, both in Python and C++

ROOT's C++ Interpreter and Python: Motivation

A more ***powerful*** C++ feature set, made available in Python automatically

Develop a ***generalized*** language interoperability solution for HEP, with modular extensions on top

Improve compiler-level interactions, ***optimizing*** resource consumption

Reduce the ***maintenance cost*** of the underlying technology behind Cppyy

In collaboration with the Compiler-Research initiative:

CppInterOp: a Clang-based C++ Interoperability library

An effort to make Cppyy more robust, closer to the compiler and more efficient

Led to a significant rework of how we perform type introspection and translate C++ language concepts into Python

CppInterOp is a solution that leverages our experience, providing building blocks for both advanced language interoperability and type introspection

This presentation aims to showcase the latest developments and potential of CppInterOp:

1. Enabling cutting edge R&D in the domain of language bindings
2. Providing better encapsulation of C++ reflection information in ROOT
3. Optimizing performance and bringing new features to Cppyy

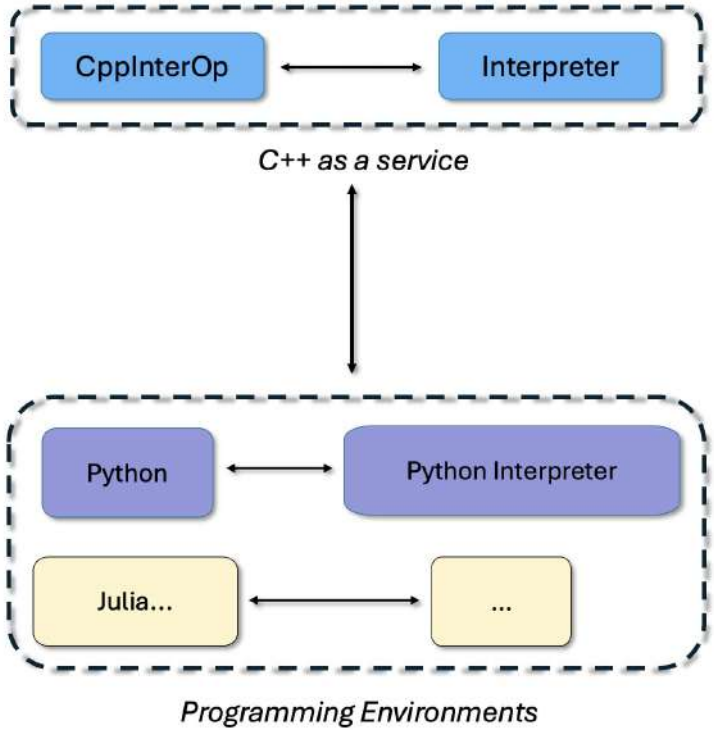
A lightweight layer on top of LLVM that provides efficient, on-demand reflection and a JIT compiler

Easily embeds Clang and LLVM as a libraries in your framework

Supports downstream tools that utilize interactive C++ by using the compiler as a service.

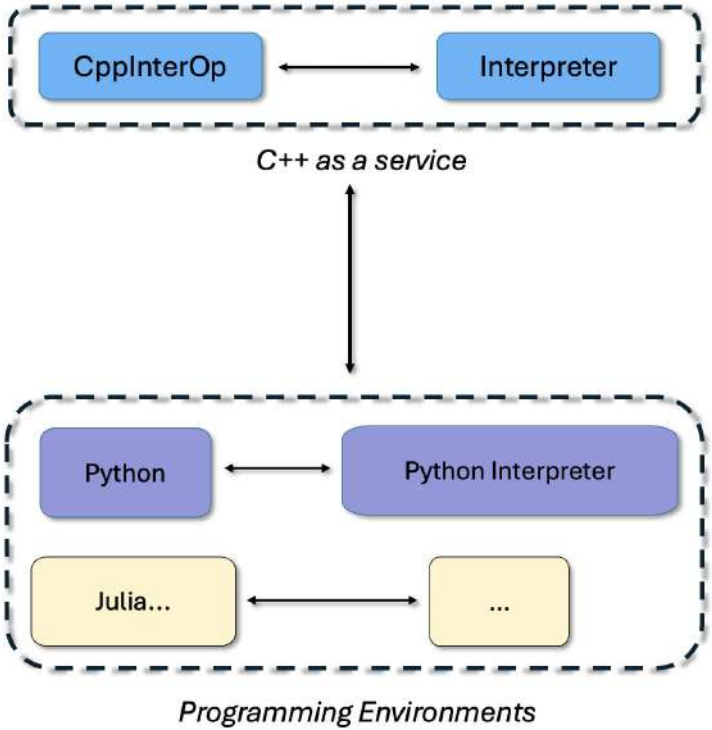
Minimal and powerful API, designed to aid non-trivial tasks, driving seamless language interoperability

Designed to work with both the Cling interpreter, and Clang-REPL, its generalisation in upstream LLVM



https://github.com/compiler-research/CppInterOp

```
Interp->process(R"(
  class TOperator{
  public:
    template<typename T>
    bool operator<(T t) { return true; }
  };
)");
Cpp::TCppScope_t TOperator = Cpp::GetNamed("TOperator");

auto* TOpCtor = Cpp::GetDefaultConstructor(TOperator);
auto FCI_TOpCtor = Cpp::MakeFunctionCallable(TOpCtor);
void* toperator = nullptr;
FCI_TOpCtor.Invoke((void*)&toperator);

EXPECT_TRUE(toperator);
```



https://github.com/compiler-research/CppInterOp

Synergy with other open-source projects

Interest in CppInterOp from the Julia community

Several contributions by a *Clang.jl* developer on the C-API

```
julia> import CppInterOp as Cpp

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

julia> Cpp.declare(I, """#include <ctime>""")
true

julia> x = Cpp.lookup(I, "clock")

julia> Cpp.name(x) == "clock"
true

julia> t1 = Cpp.evaluate(I, "clock()"); result = Ref{Clong}
(C_NULL); Cpp.invoke(x, result);t2 = Cpp.evaluate(I, "clock()")
true

julia> t1 < result[] < t2
true
```

CppInterOp.jl

Recipe to build CppInterOp as a Julia package:

https://github.com/JuliaPackaging/Yggdrasil/tree/master/L/libCppInterOp

Synergy with other open-source projects

jank

https://jank-lang.org/

Dialect of Clojure based on LLVM JIT

Extends CppInterOp for C++ Interoperability

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

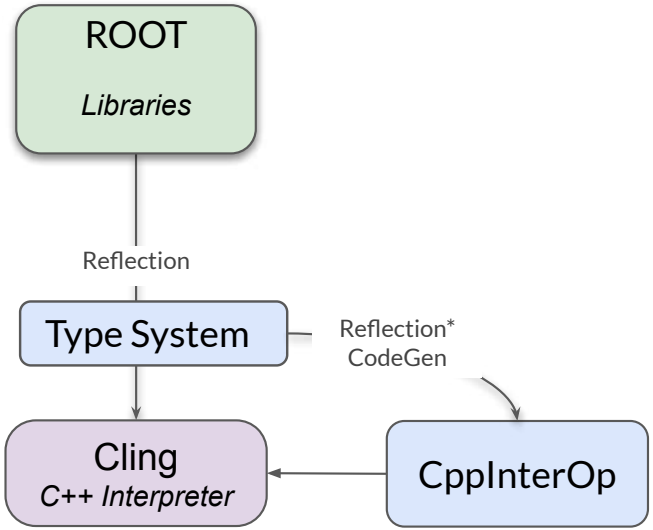
```
(cpp/raw "namespace
jank::cpp::constructor::complex::pass_aliased
         {
           struct {
             int padding{};
             int a{ 333 };
           };
           using T = foo;
         }")
(let* [foo
(cpp/jank.cpp.constructor.complex.pass_aliased.T.)]
  (if (= 333 (cpp/.-a foo))
    :success))
```

Integration with ROOT

CppInterOp has been **integrated** into ROOT from v6.36

Incrementally abstracts parts of the type system

- Offloads ***some*** reflection
- Leverages code generation facilities for function wrappers



Redesigned Cppyy

Cppyy re-engineered to be based on **CppInterOp**

Works standalone today!

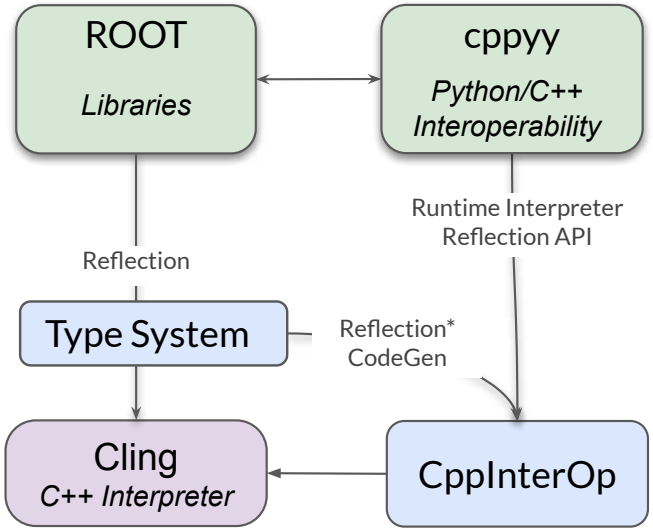
Passes ~95% of the Cppyy test cases

Currently being integrated in ROOT

Fewer dependencies, more performant

Adheres to Clang for typing, conversion and overload resolution rules - less bugs

More ***language features*** from C++ can be used in Python *Eg. templated callbacks, lambdas, and more*



Ongoing Integration

Features enabled by the new cppyy

- *Templated Callbacks*: Users can now pass templated functions as a callback to higher order functions

In Development

```python
data = ROOT.vector[float]([1.0, 2.0, 3.0])
out = ROOT.map(data, gbl.foo[float, int], 3.0, 0)
print(data) # { 1.00000f, 2.00000f, 3.00000f }
print(out)  # { 4.00000f, 5.00000f, 6.00000f }
```

```cpp
template <typename T1, typename T2>
double foo(double in, T1 a, T2 b) {
   return in + a + b; };
template<typename I, typename F, typename ... Args>
I map(I iterable, F callable, Args&& ... args) { ...
}
```

- *C++ lambdas*: Ability to access and use C++ lambdas

```python
assert ROOT.get_add_fn(5)(15) == 20
```

```cpp
auto get_add_fn(int x) {
 return [x](int y) { return x + y; };
}
```

- *Better support for std::tuple*: Tuple indexing and unpacking

```python
t = ROOT.std.make_tuple(1, "2", 5.0)
assert len(t) == 3

a, b, c = t
assert a == 1
assert b == "2"
assert c == 5.0
```

- *Template Instantiation based on Python Type Hints*

```python
def callback(x: int, y: float) -> float:
   return x + y

ROOT.callme(callback, 123, 321.5) # 444.5
```

```cpp
template <typename R, typename ... U,
          typename ... A>
R callme(R (*f)(U ... ), A && ... args) {
 return f(args ... );
}
```

Better Integration with Developer Tools

- Python Type Hints
- Auto Completes
- Documentation on Hover
- For ROOT types and configurable for custom user defined types

In Development

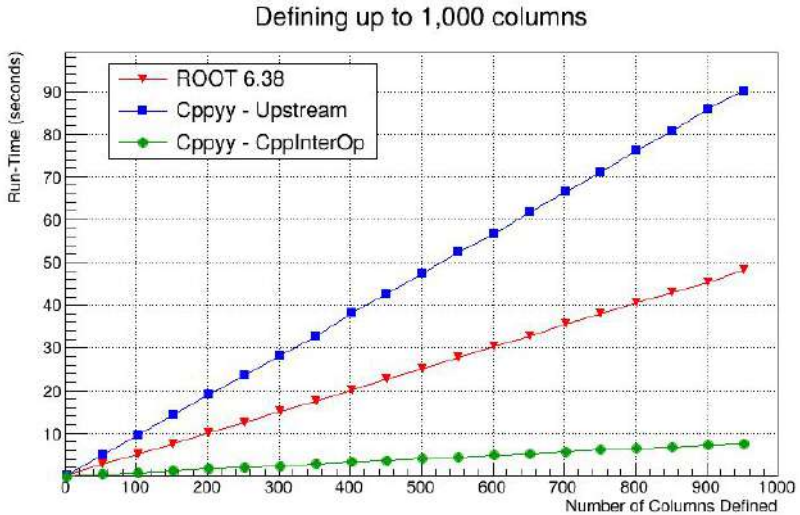
Better Integration with Developer Tools

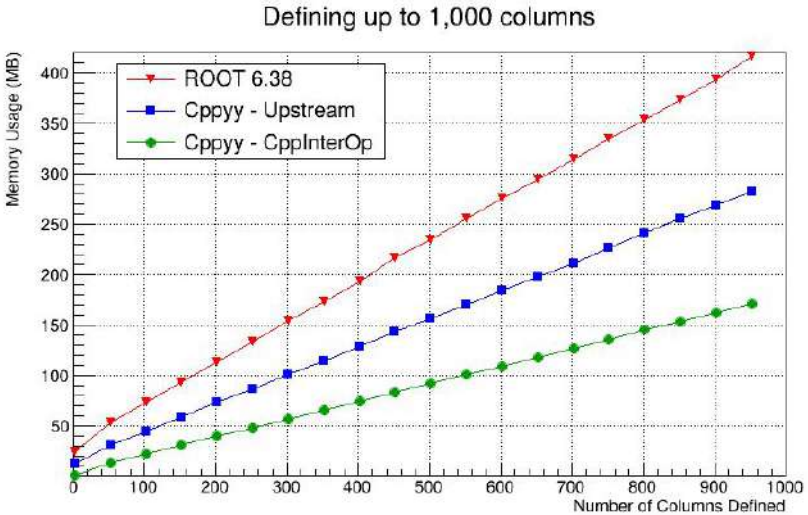
- Parse source code to extract types and doxygen comments.
- Generate `.pyi` interface files using type info and docs comments.
- Interface files packaged with ROOT for ROOT types.
- Dynamically generated for user defined types.

```python
import ROOT

ROOT.gInterpreter.Declare(r"""
struct UserDefined { ... };
""")

ROOT.generate_interface(ROOT.UserDefined, output="typings/")
```

- We compare the new Cppyy based on CppInterOp against Upstream and ROOT 6.38.
- The benchmark measures the call-overhead instantiating a lightweight data frame class modelled after RDataFrame, defining up to 1000 columns.

```
df.Define("c1", f1).Define("c2", f2)...
```





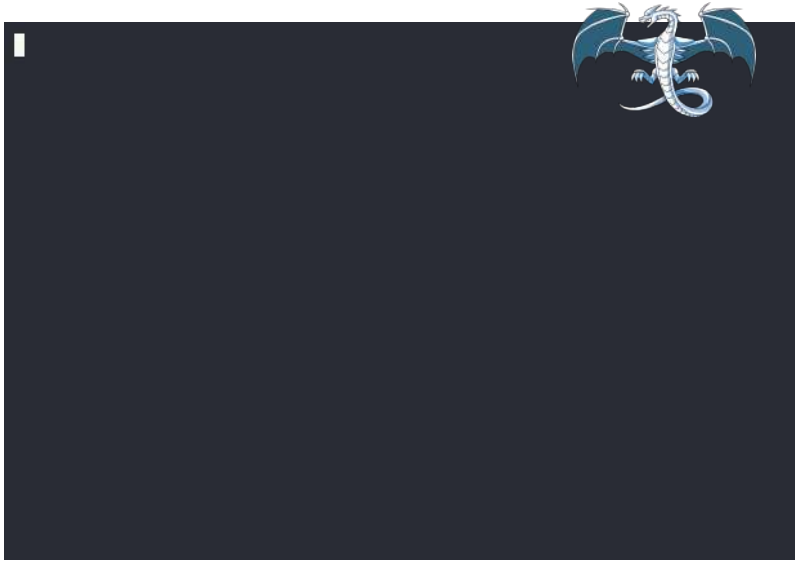
Summary

- *The development of CppInterOp, a new lightweight library on top of LLVM, driving improved reflection and language bindings*
- *Its adoption within ROOT, improving maintainability, stability and performance*
- *Redesigned Cppyy based on CppInterOp, opening up new features, driving down run time and memory expenses. Currently being integrated into ROOT*
- *Synergy with other open-source projects, powering interoperability with other languages*
- *Participating in the broader programming language and compiler community*

ROOT's interactive C++ Interpreter

Built on the **LLVM/Clang** compiler framework

Allows for the runtime execution of C++ code



Backup: Cppyy Upstream

Cppyy upstream is currently based on ROOT 6.32.8 (Dec 2024)

cppyy.readthedocs.io

```
>>> 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
>>>
```