spcl.ethz.ch @spcl @spcl_eth

CSCS ETH zürich

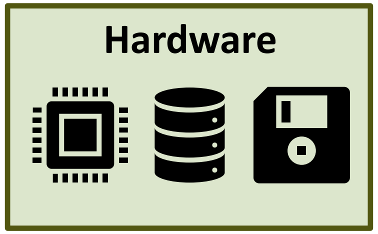
Marcin Copik, Lukas Möller, Alexandru Calotoiu, Torsten Hoefler

Cppless: Single-Source and High-Performance Serverless Programming in C++

HiPEAC 2026
Cracow, Poland

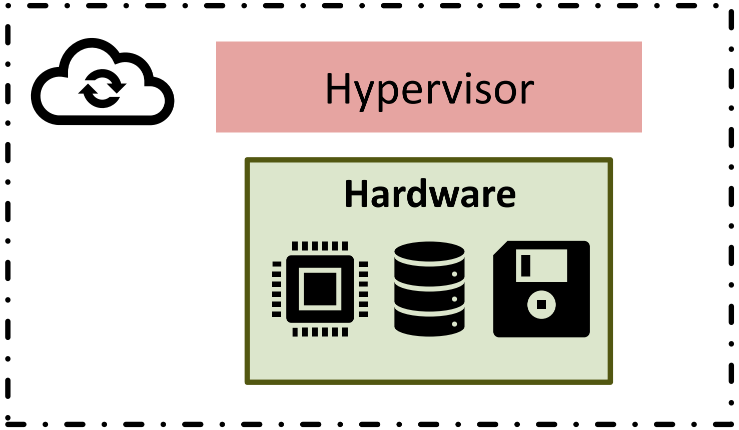
Cloud and Serverless

Cloud and Serverless

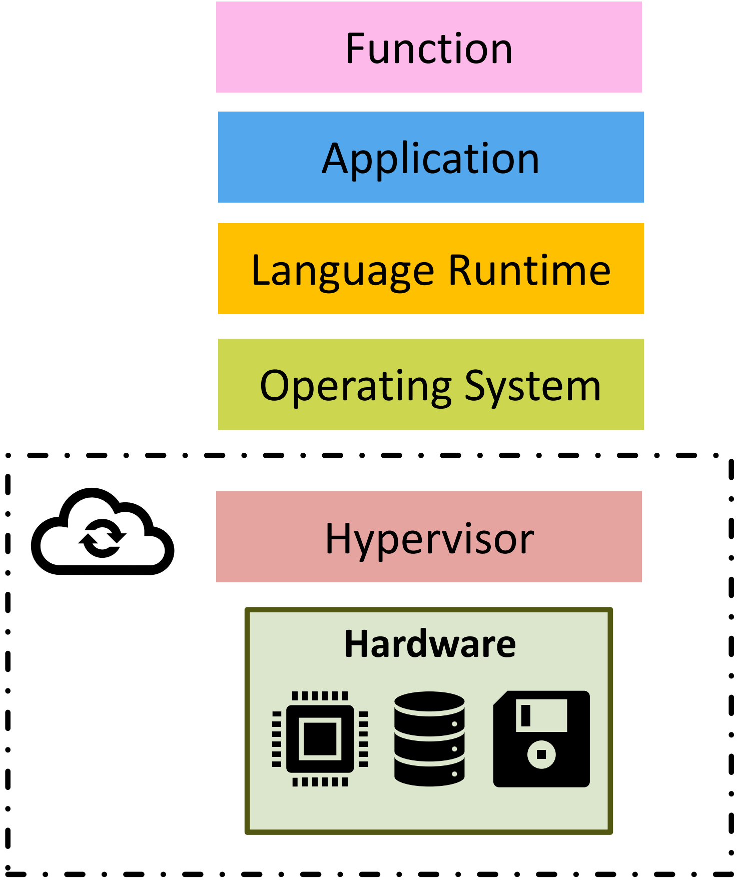


Hardware

Cloud and Serverless

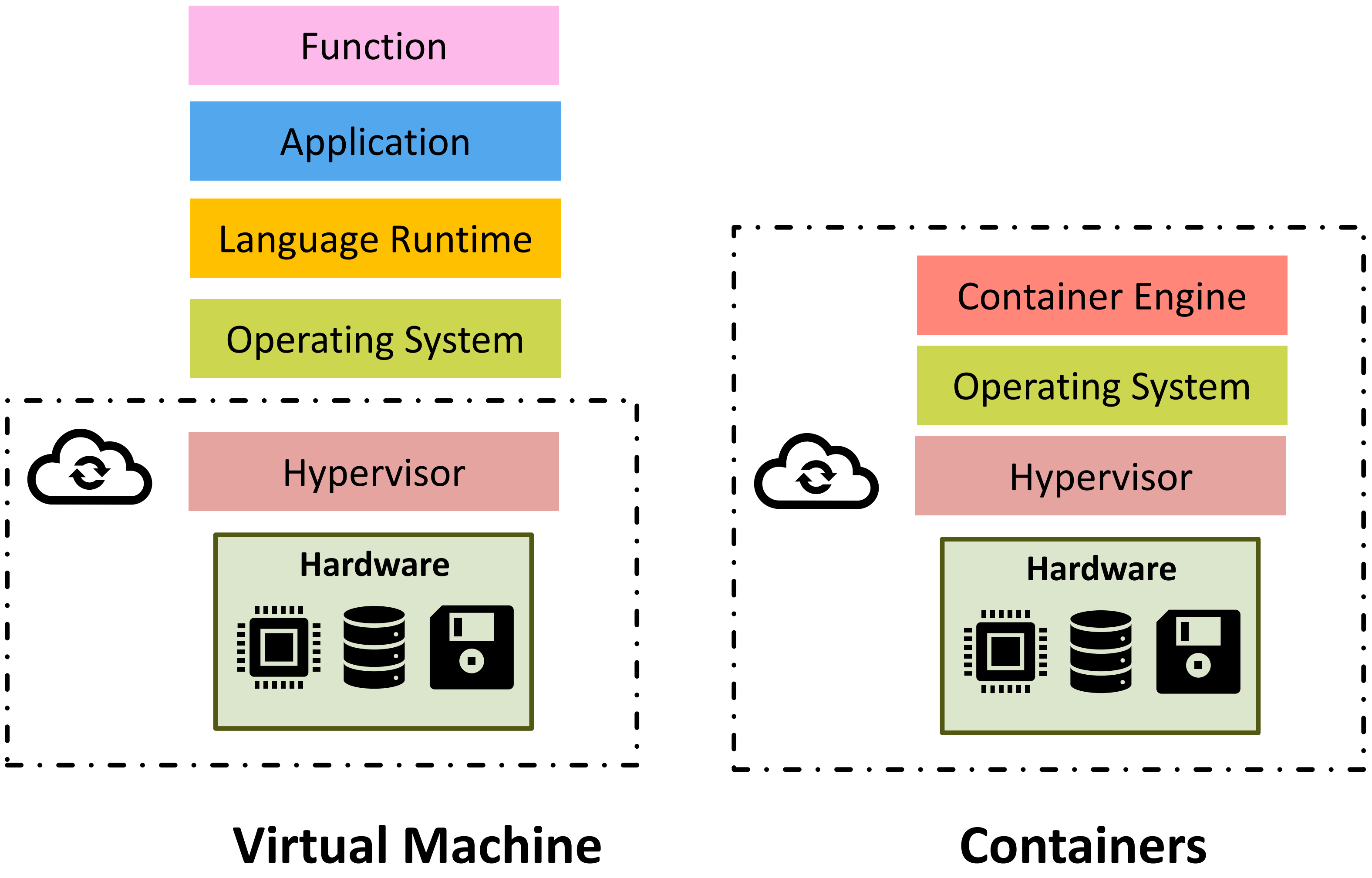


Cloud and Serverless


Function
Application
Language Runtime
Operating System
Hypervisor
Hardware

Virtual Machine

Cloud and Serverless



Function

Application

Language Runtime

Operating System

Hypervisor

Hardware

Virtual Machine

Container Engine

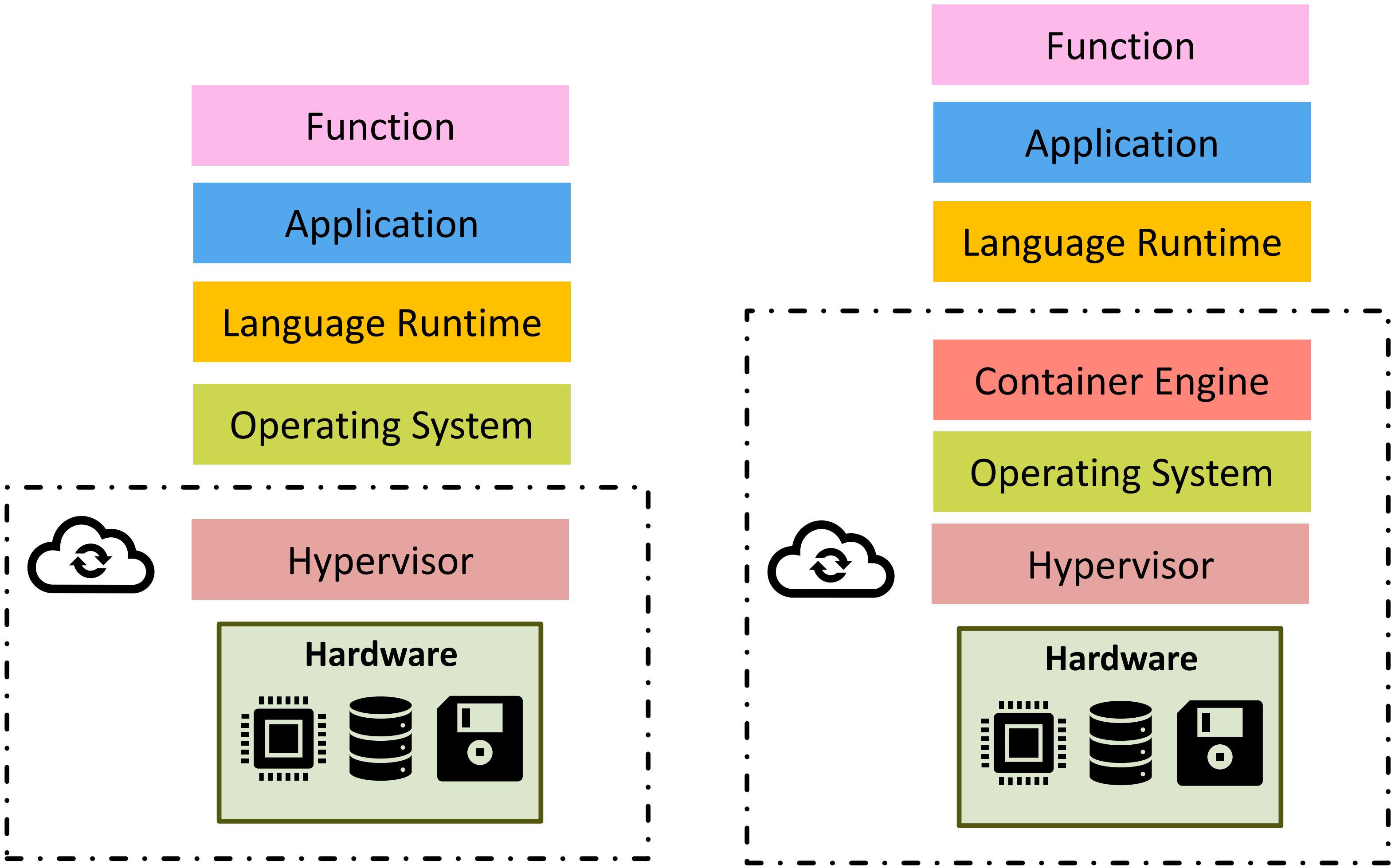
Operating System

Hypervisor

Hardware

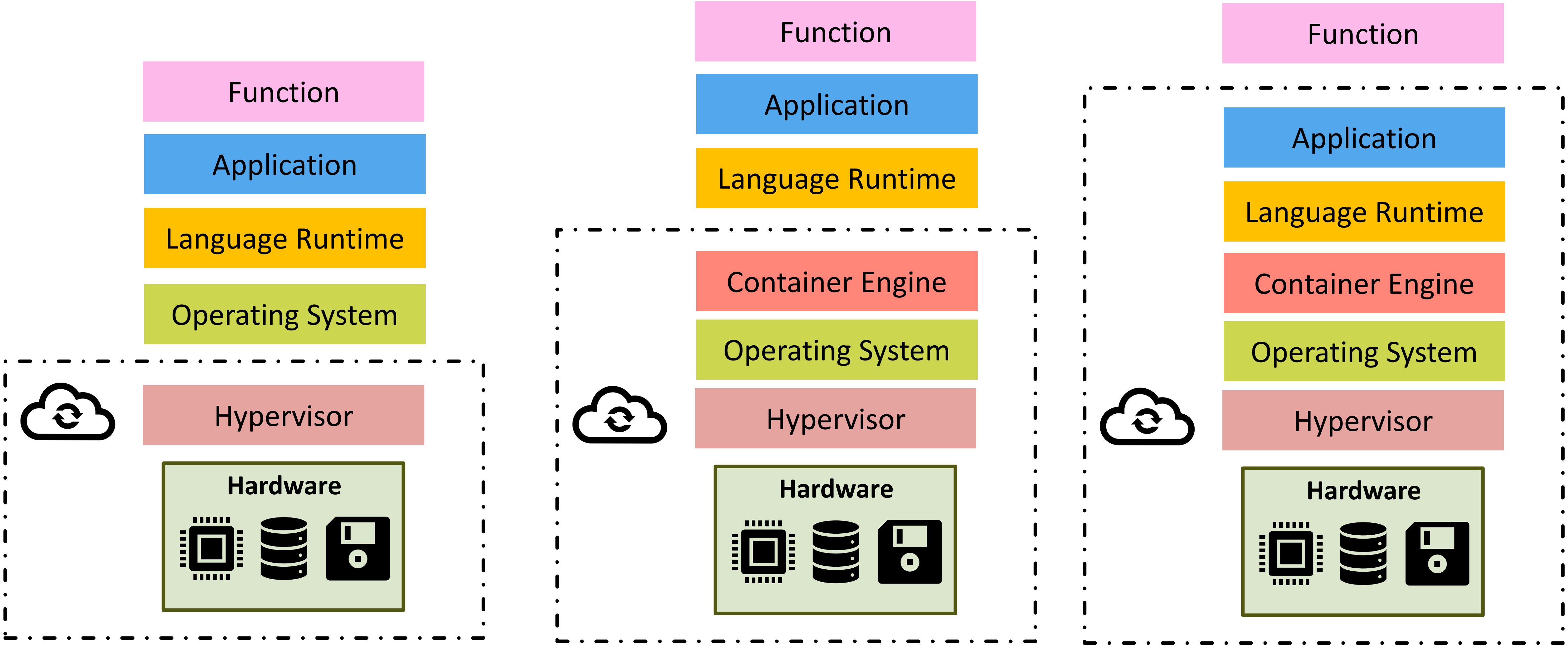
Containers

Cloud and Serverless



Virtual Machine **Containers**

Cloud and Serverless

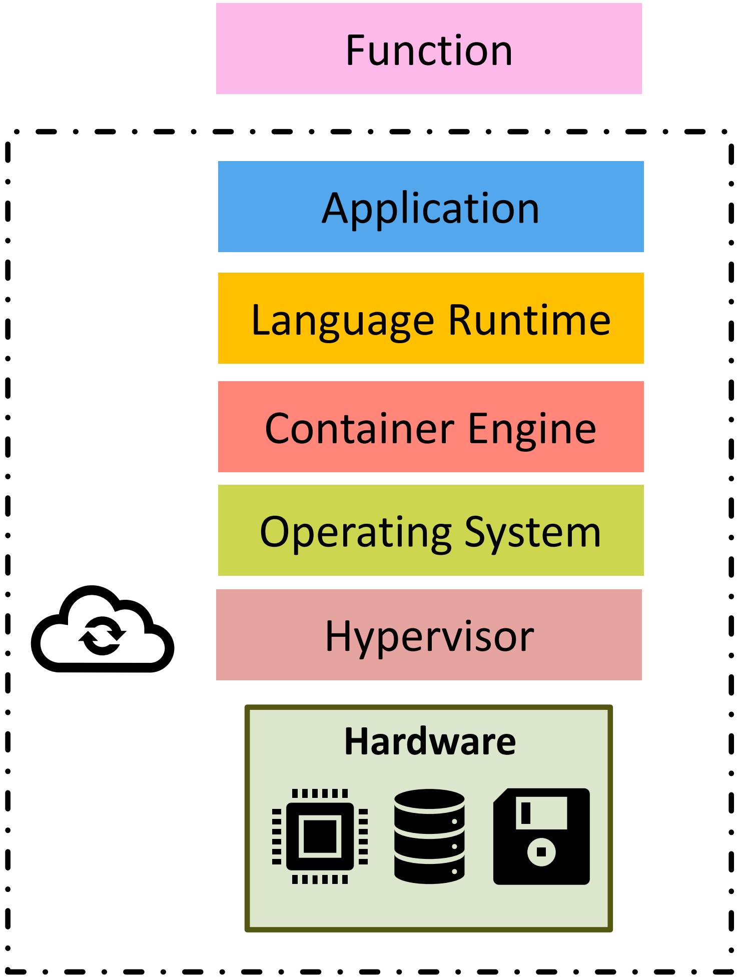


Virtual Machine

Containers

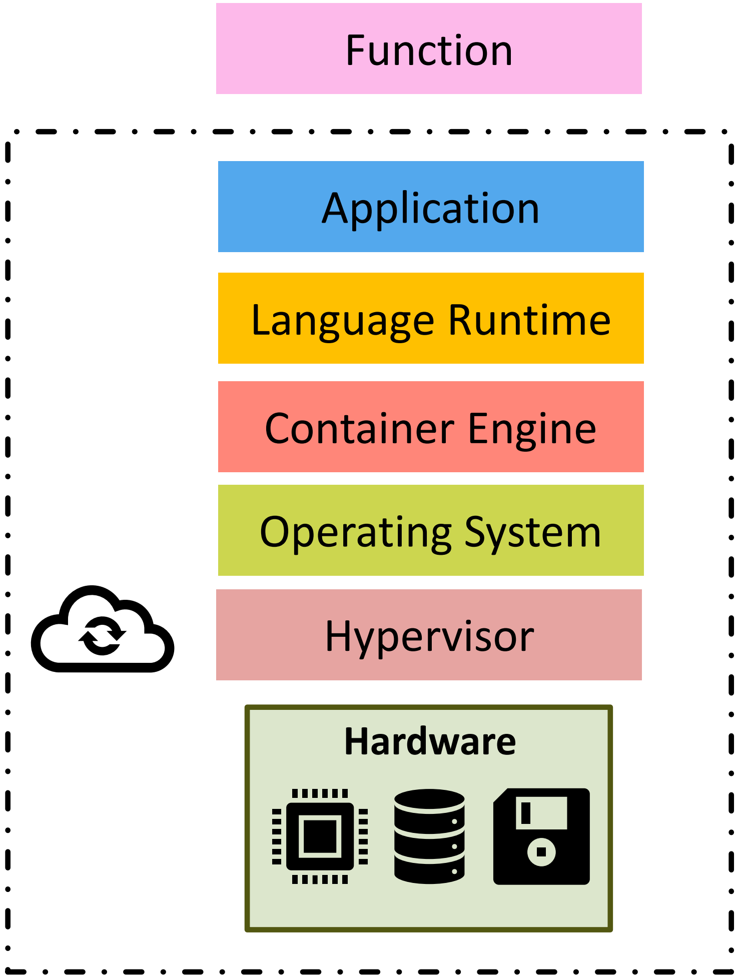
Functions

Cloud and Serverless



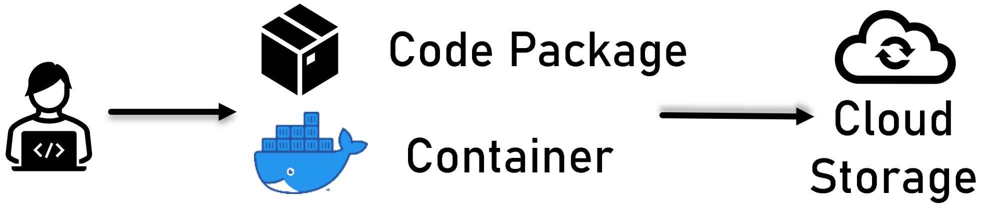
Functions

Cloud and Serverless

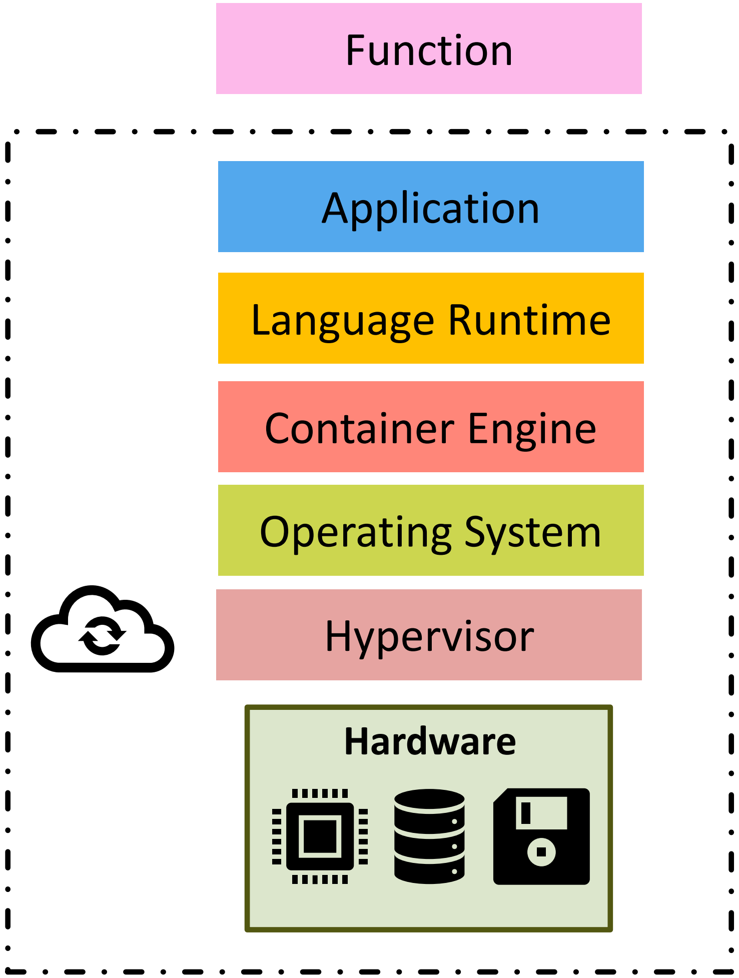


Functions

Compilation: deploying functions to the cloud.

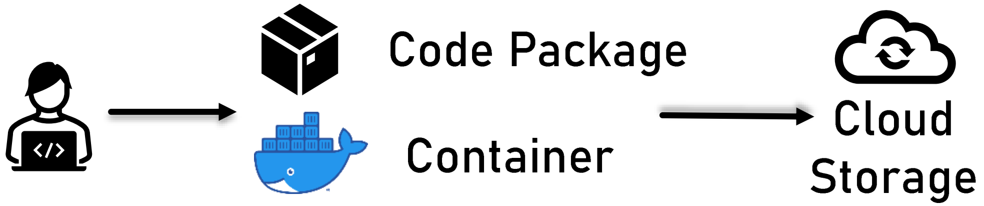


Cloud and Serverless

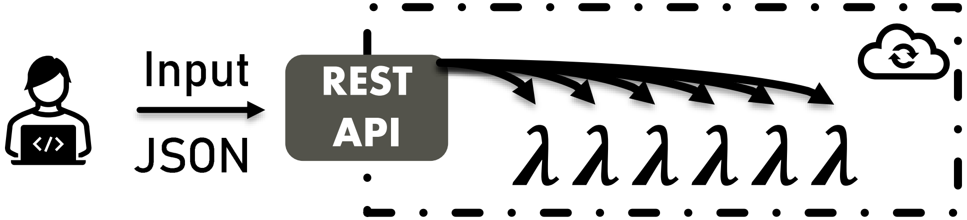


Functions

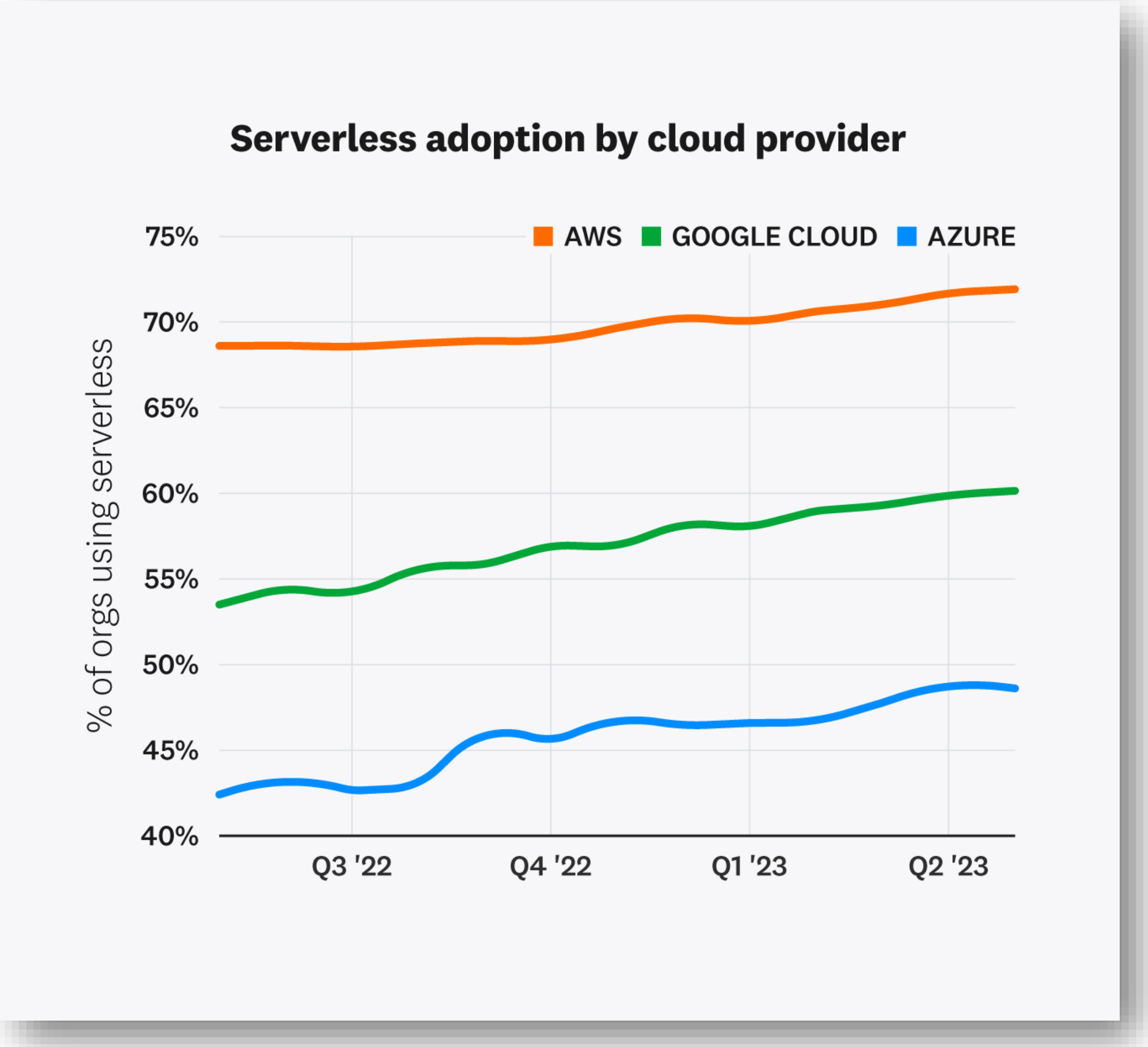
Compilation: deploying functions to the cloud.



Runtime: invoking existing functions.

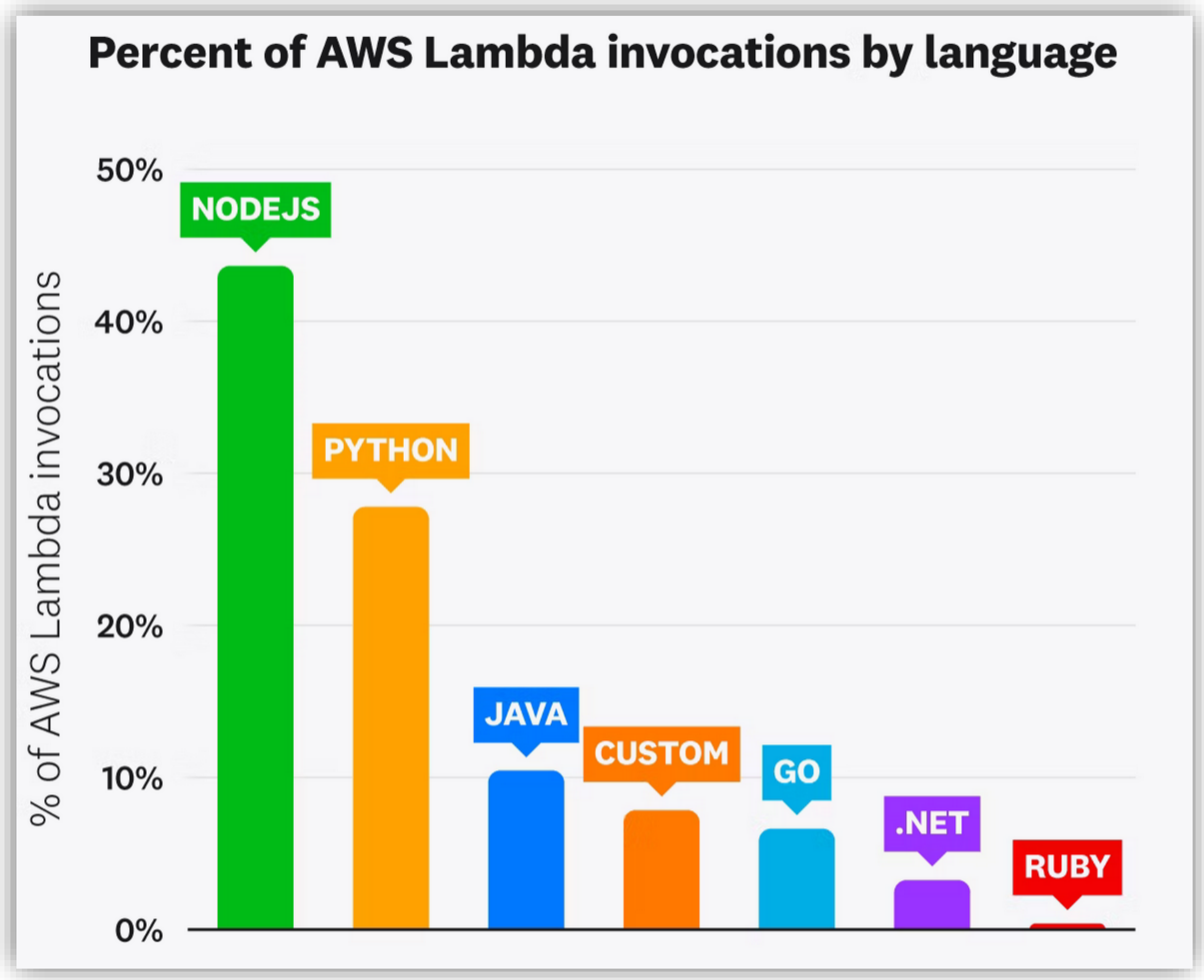


Serverless – Beyond Node.js and Python



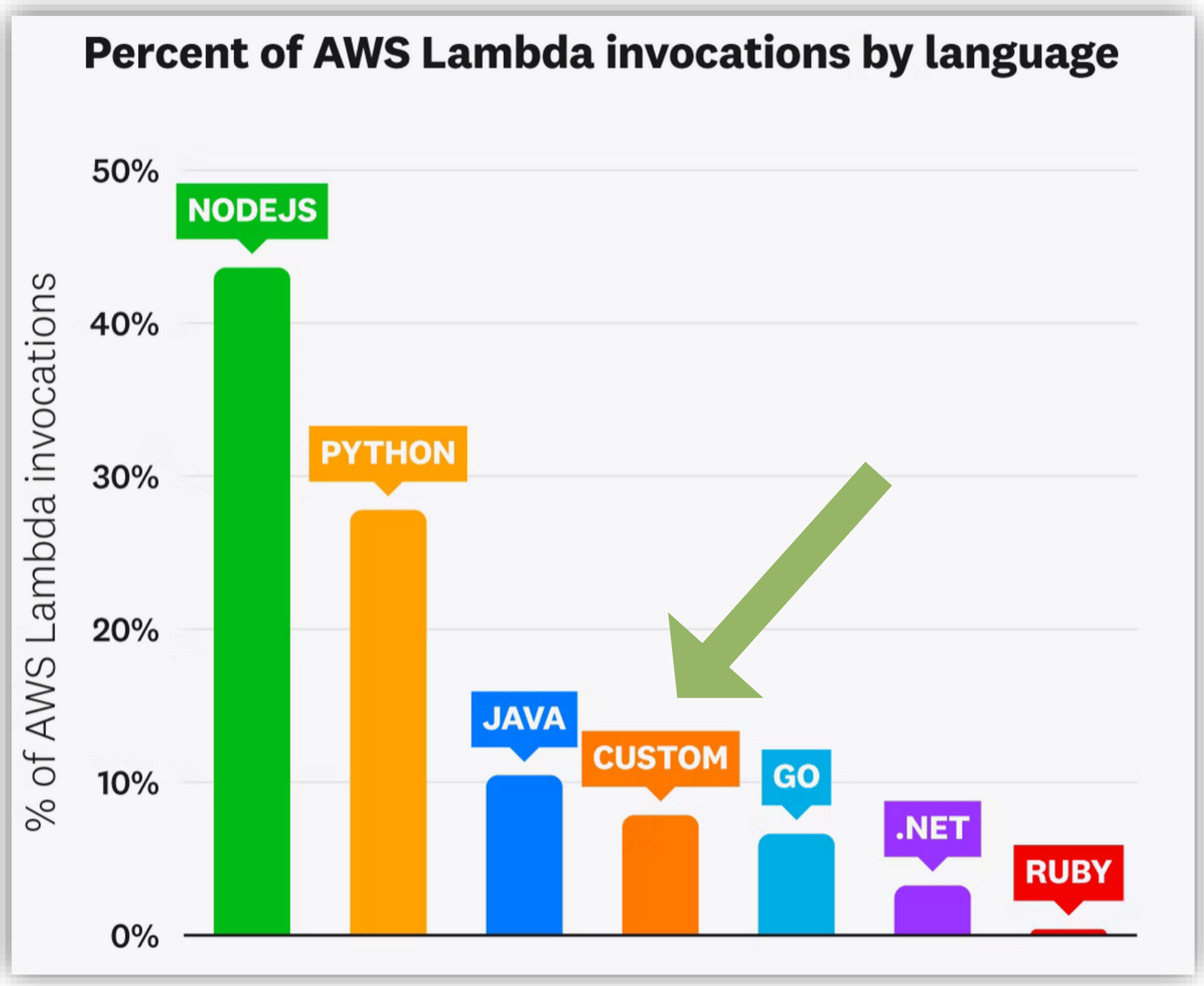
Source: DataDog, “State of Serverless 2023”, https://www.datadoghq.com/state-of-serverless/

Serverless – Beyond Node.js and Python



Source: DataDog, "State of Serverless 2023", https://www.datadoghq.com/state-of-serverless/

Serverless – Beyond Node.js and Python

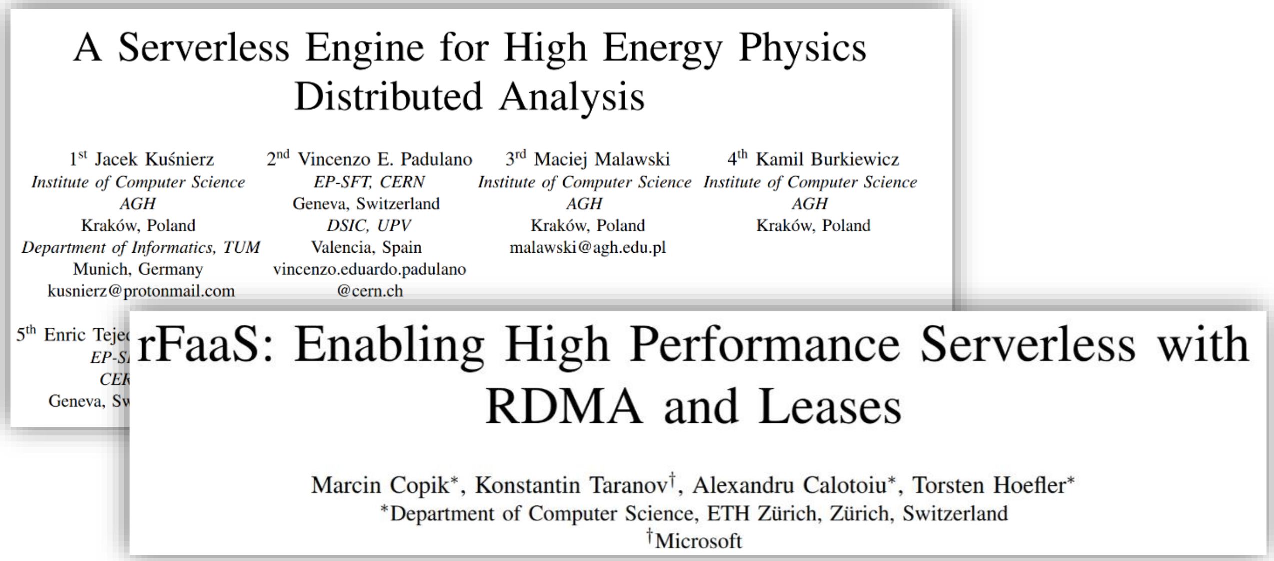


Source: DataDog, “State of Serverless 2023”, https://www.datadoghq.com/state-of-serverless/

Why C++ for Serverless?

Why C++ for Serverless?

Scientific Computing



A Serverless Engine for High Energy Physics Distributed Analysis

1st Jacek Kuśnierz
Institute of Computer Science
AGH
Kraków, Poland
Department of Informatics, TUM
Munich, Germany
kusnierz@protonmail.com

2nd Vincenzo E. Padulano
EP-SFT, CERN
Geneva, Switzerland
DSIC, UPV
Valencia, Spain
vincenzo.eduardo.padulano
@cern.ch

3rd Maciej Malawski
Institute of Computer Science
AGH
Kraków, Poland
malawski@agh.edu.pl

4th Kamil Burkiewicz
Institute of Computer Science
AGH
Kraków, Poland

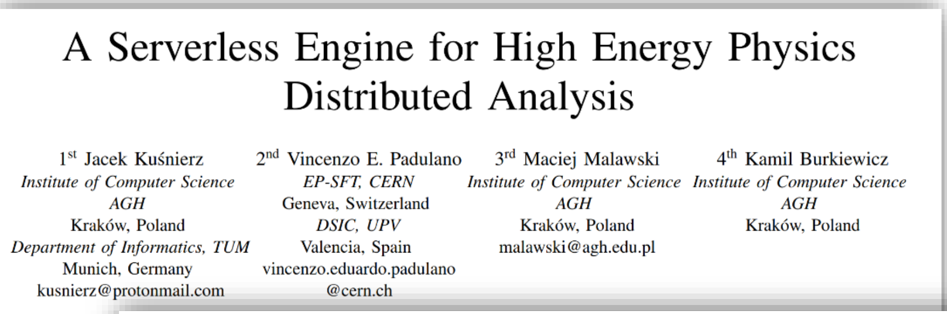
5th Enric Tejed
EP-S
CER
Geneva, Sw

rFaaS: Enabling High Performance Serverless with RDMA and Leases

Marcin Copik*, Konstantin Taranov†, Alexandru Calotoiu*, Torsten Hoefler*
*Department of Computer Science, ETH Zürich, Zürich, Switzerland
†Microsoft

Why C++ for Serverless?

Scientific Computing



A Serverless Engine for High Energy Physics Distributed Analysis

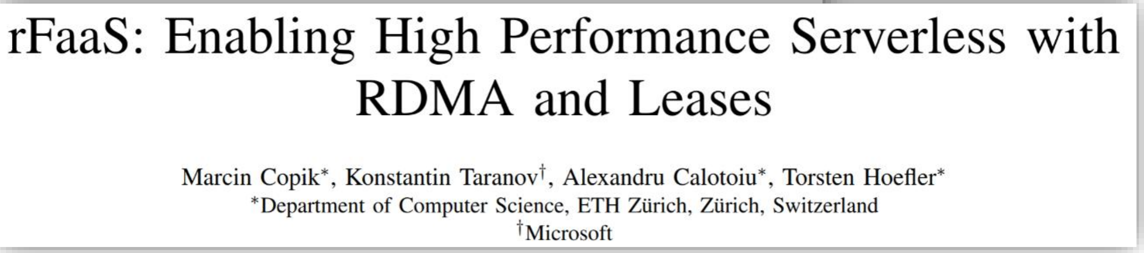
1st Jacek Kuśnierz
Institute of Computer Science
AGH
Kraków, Poland
Department of Informatics, TUM
Munich, Germany
kusnierz@protonmail.com

2nd Vincenzo E. Padulano
EP-SFT, CERN
Geneva, Switzerland
DSIC, UPV
Valencia, Spain
vincenzo.eduardo.padulano
@cern.ch

3rd Maciej Malawski
Institute of Computer Science
AGH
Kraków, Poland
malawski@agh.edu.pl

4th Kamil Burkiewicz
Institute of Computer Science
AGH
Kraków, Poland

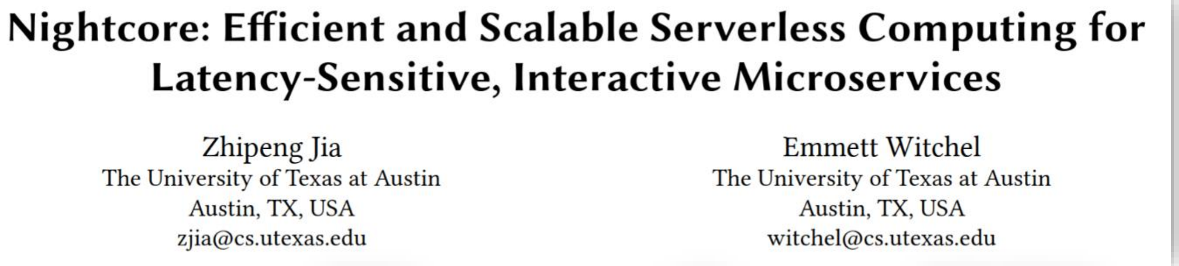
5th Enric Teje
EP-S
CER
Geneva, Sw



rFaaS: Enabling High Performance Serverless with RDMA and Leases

Marcin Copik*, Konstantin Taranov†, Alexandru Calotoiu*, Torsten Hoefler*
*Department of Computer Science, ETH Zürich, Zürich, Switzerland
†Microsoft

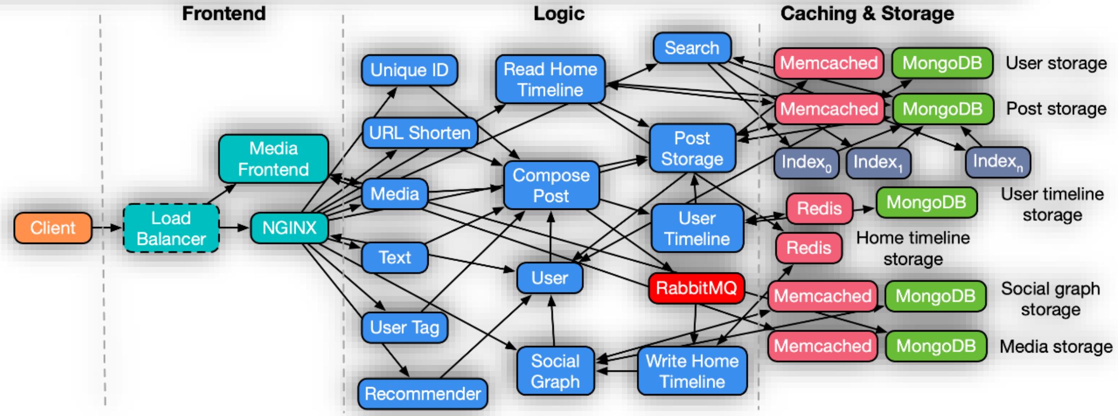
Microservices



Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices

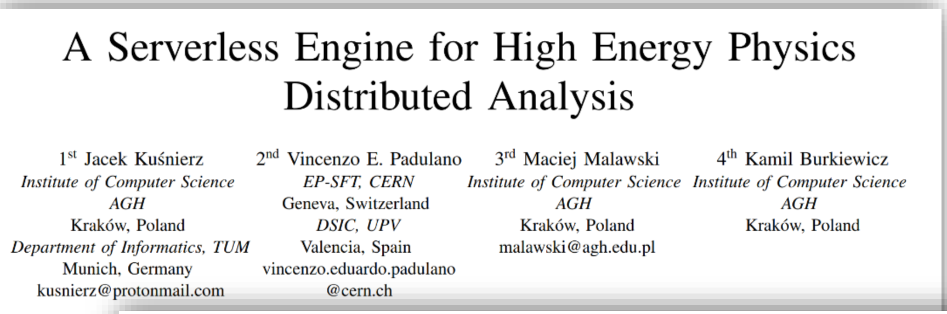
Zhipeng Jia
The University of Texas at Austin
Austin, TX, USA
zjia@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
Austin, TX, USA
witchel@cs.utexas.edu



Why C++ for Serverless?

Scientific Computing



A Serverless Engine for High Energy Physics Distributed Analysis

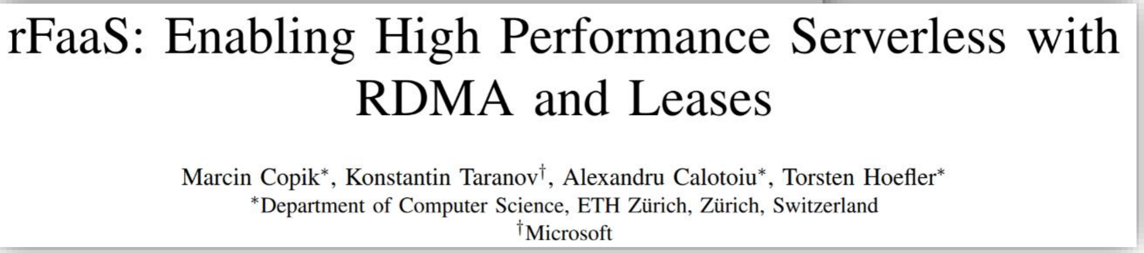
1st Jacek Kuśnierz
Institute of Computer Science
AGH
Kraków, Poland
Department of Informatics, TUM
Munich, Germany
kusnierz@protonmail.com

2nd Vincenzo E. Padulano
EP-SFT, CERN
Geneva, Switzerland
DSIC, UPV
Valencia, Spain
vincenzo.eduardo.padulano@cern.ch

3rd Maciej Malawski
Institute of Computer Science
AGH
Kraków, Poland
malawski@agh.edu.pl

4th Kamil Burkiewicz
Institute of Computer Science
AGH
Kraków, Poland

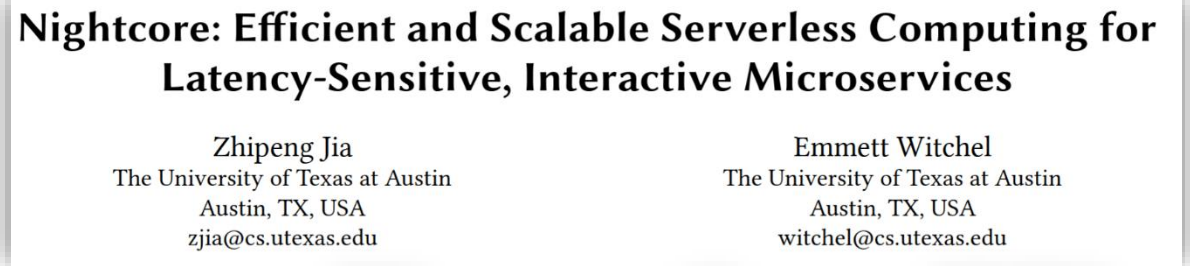
5th Enric Teje
EP-S
CER
Geneva, Sw



rFaaS: Enabling High Performance Serverless with RDMA and Leases

Marcin Copik*, Konstantin Taranov†, Alexandru Calotoiu*, Torsten Hoefler*
*Department of Computer Science, ETH Zürich, Zürich, Switzerland
†Microsoft

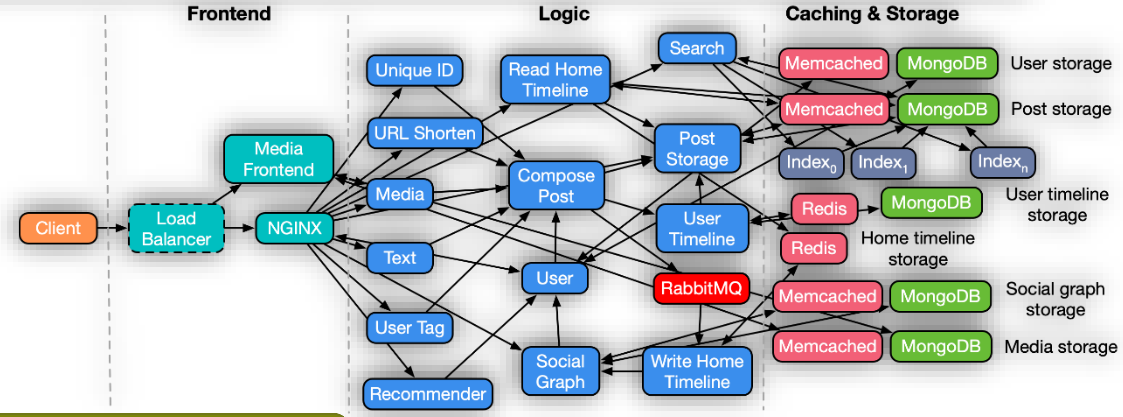
Microservices



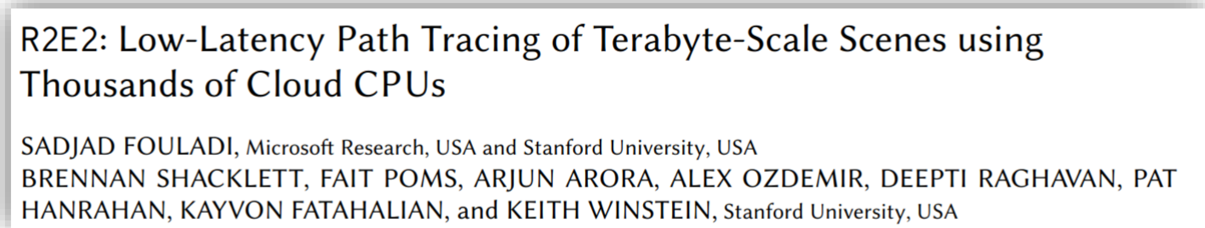
Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices

Zhipeng Jia
The University of Texas at Austin
Austin, TX, USA
zjia@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
Austin, TX, USA
witchel@cs.utexas.edu



High-Performance Applications



R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes using Thousands of Cloud CPUs

SADJAD FOULADI, Microsoft Research, USA and Stanford University, USA
BRENNAN SHACKLETT, FAIT POMS, ARJUN ARORA, ALEX OZDEMIR, DEEPTI RAGHAVAN, PAT HANRAHAN, KAYVON FATAHALIAN, and KEITH WINSTEIN, Stanford University, USA

AMD ROCm

Why C++ for Serverless?

Scientific Computing

A Serverless Engine for High Energy Physics Distributed Analysis

1st Jacek Kuśnierz, Institute of Computer Science, AGH, Kraków, Poland
2nd Vincenzo E. Padulano, EP-SFT, CERN, Geneva, Switzerland, DSIC, UPV
3rd Maciej Malawski, Institute of Computer Science, AGH, Kraków, Poland
4th Kamil Burkiewicz, Institute of Computer Science, AGH, Kraków, Poland

Microservices

Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices

Zhipeng Jia
The University of Texas at Austin
Austin, TX, USA
zjia@cs.utexas.edu

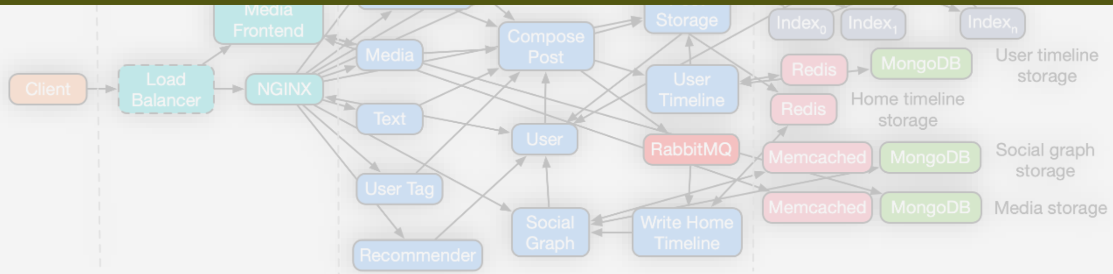
Emmett Witchel
The University of Texas at Austin
Austin, TX, USA
witchel@cs.utexas.edu

Can Serverless Functions Become First-Class Citizens?

Geneva, Sw

RDMA and Leases

Marcin Copik[*], Konstantin Taranov[†], Alexandru Calotoiu[*], Torsten Hoefler[*]
[*]Department of Computer Science, ETH Zürich, Zürich, Switzerland
[†]Microsoft



High-Performance Applications

R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes using Thousands of Cloud CPUs

SADJAD FOULADI, Microsoft Research, USA and Stanford University, USA
BRENNAN SHACKLETT, FAIT POMS, ARJUN ARORA, ALEX OZDEMIR, DEEPTI RAGHAVAN, PAT HANRAHAN, KAYVON FATAHALIAN, and KEITH WINSTEIN, Stanford University, USA

AMD ROCm

Separate Codebase vs Single-Source Serverless

BUILD

RUN

Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow

BUILD

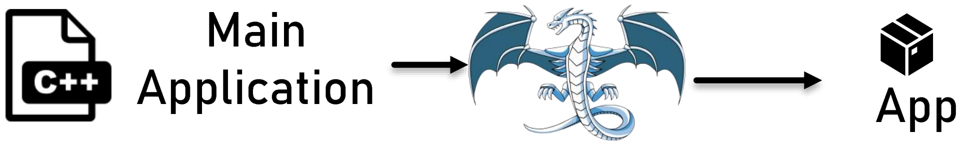
RUN

Separate Codebase vs Single-Source Serverless

BUILD

RUN

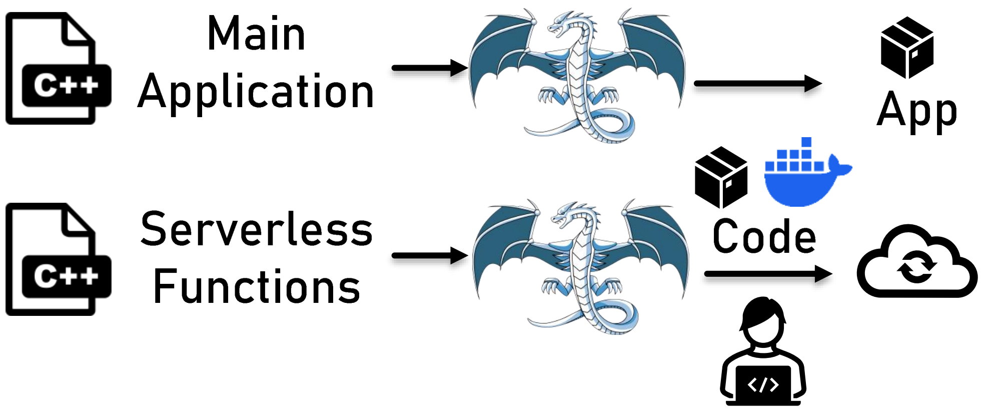
Standard Serverless Workflow



Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow

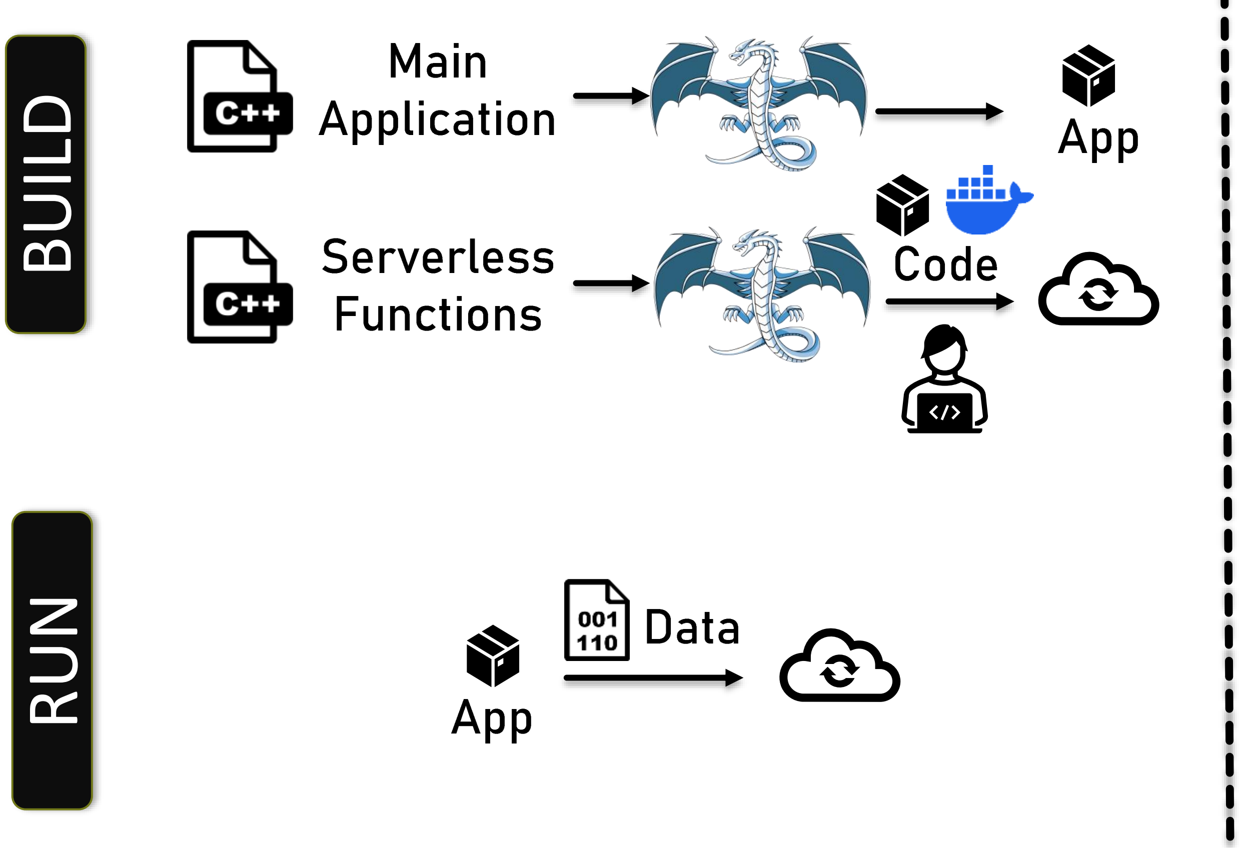
BUILD



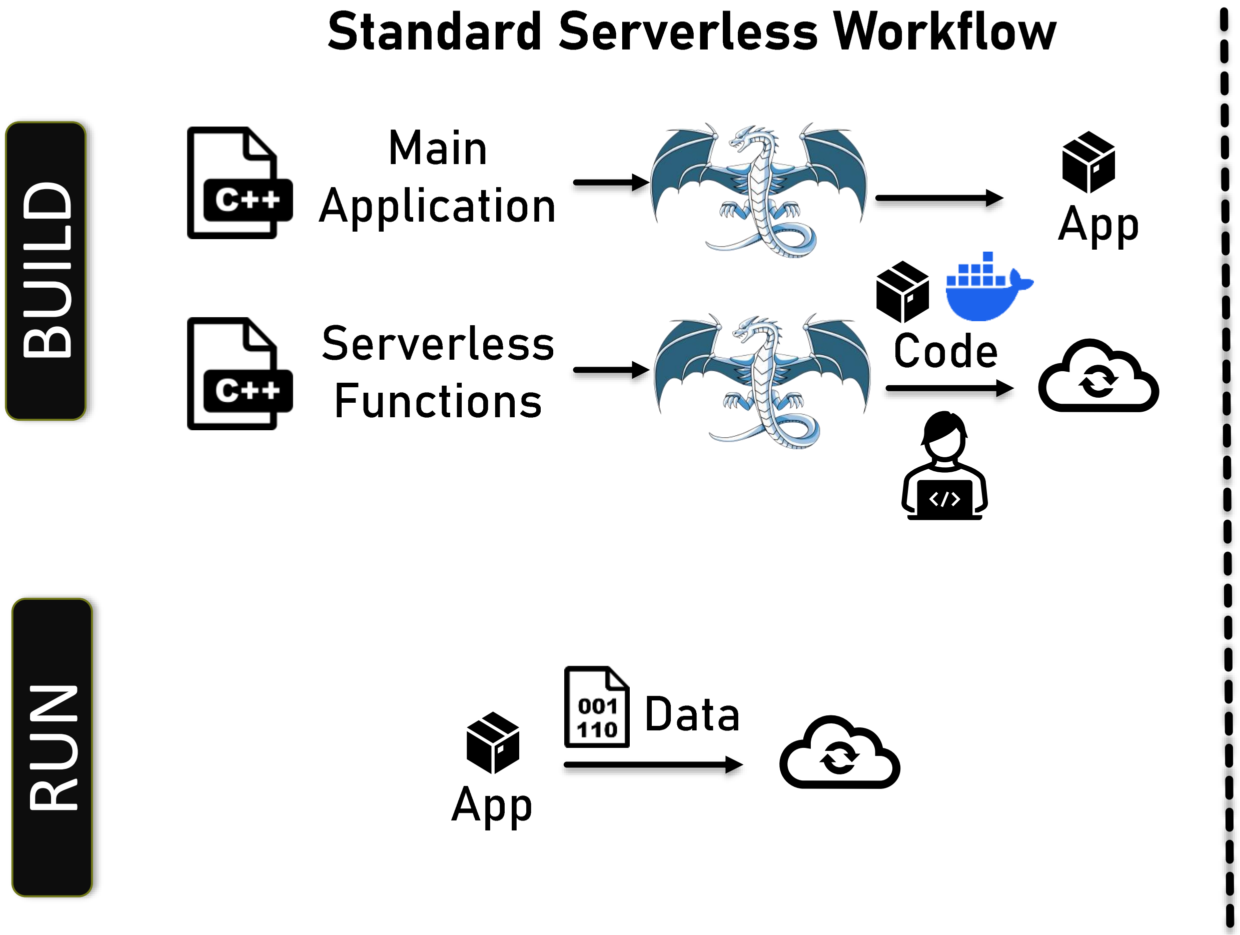
RUN

Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow



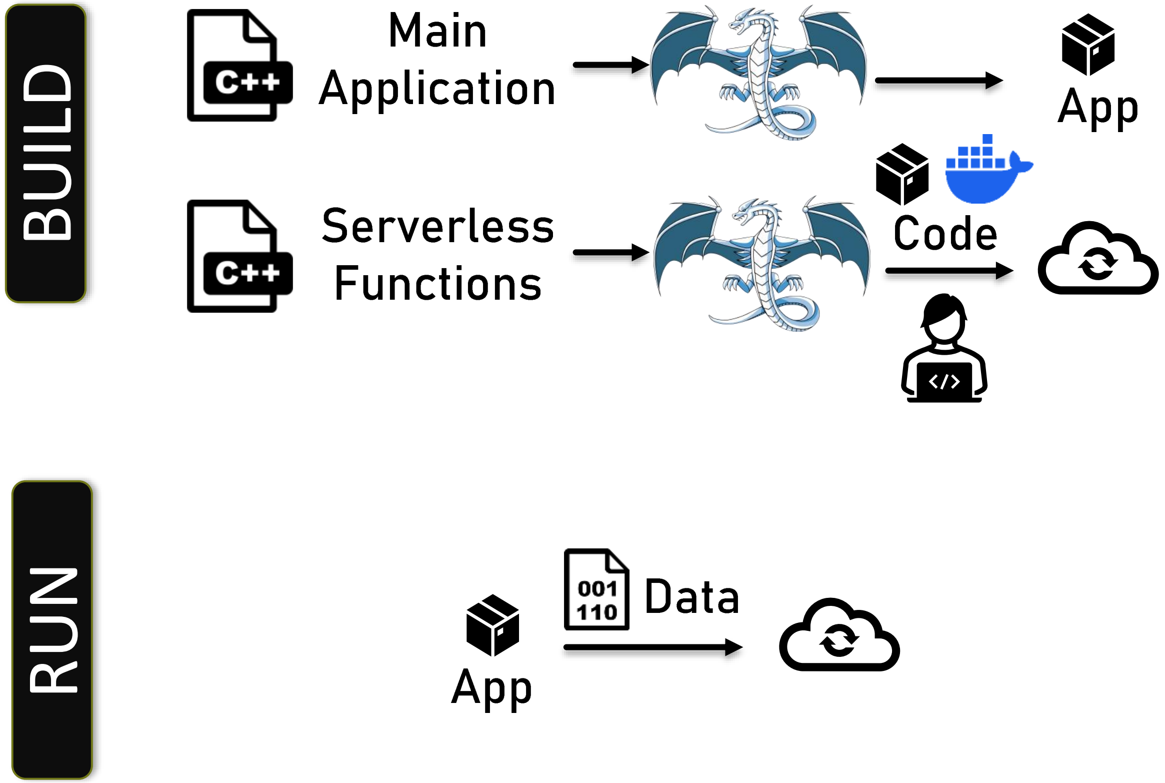
Separate Codebase vs Single-Source Serverless



✖ Manual code separation, error-prone

Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow



BUILD

Main Application

App

Serverless Functions

Code

RUN

App

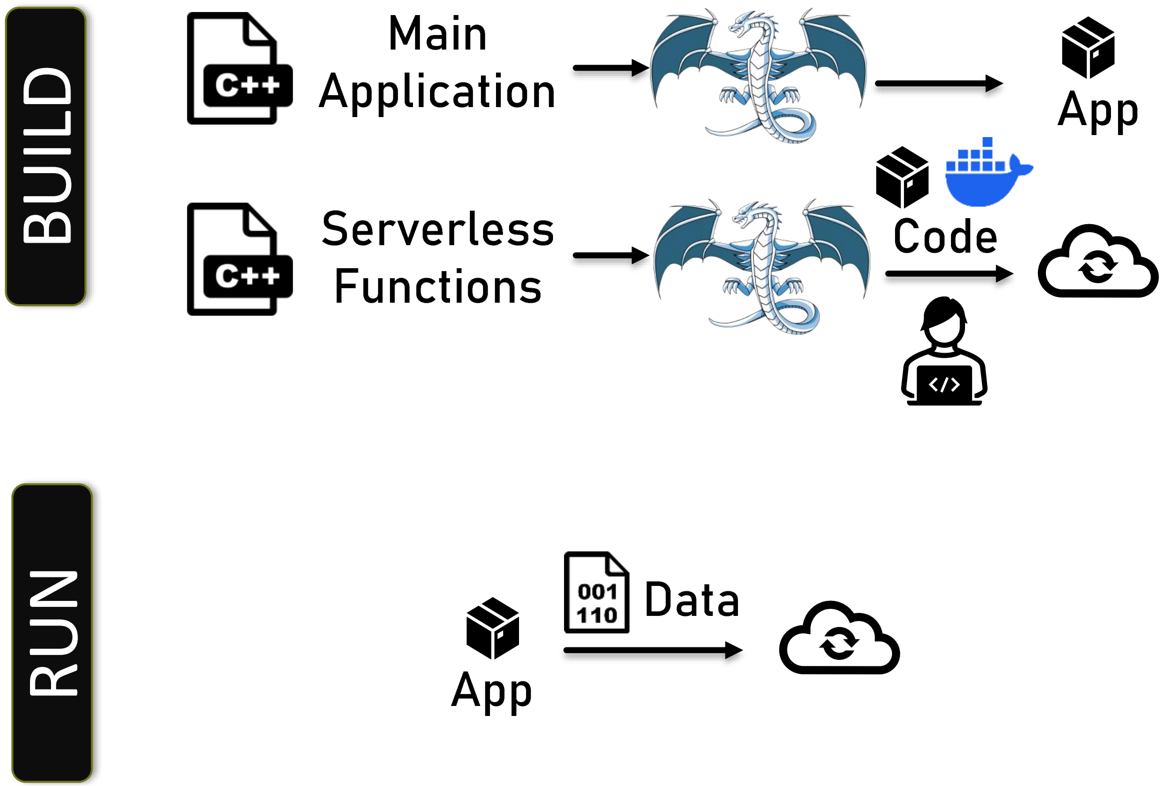
Data

Dynamic Dispatch (Python, Java, Bash)

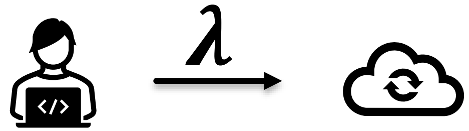
✖ Manual code separation, error-prone

Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow



Dynamic Dispatch (Python, Java, Bash)



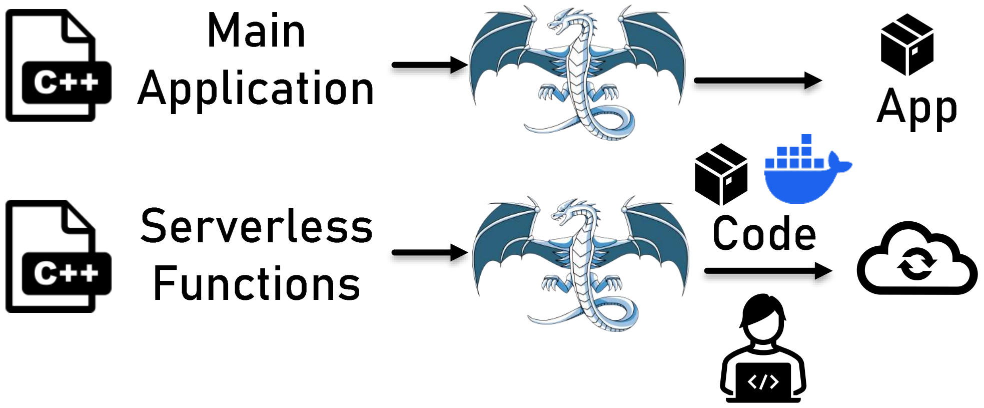
Define and configure generic functions.

✗ Manual code separation, error-prone

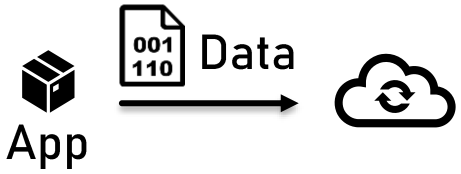
Separate Codebase vs Single-Source Serverless

Standard Serverless Workflow

BUILD

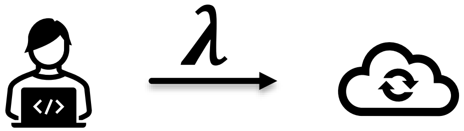


RUN

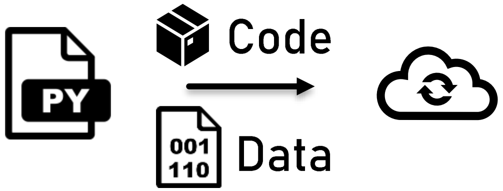


✖ Manual code separation, error-prone

Dynamic Dispatch (Python, Java, Bash)

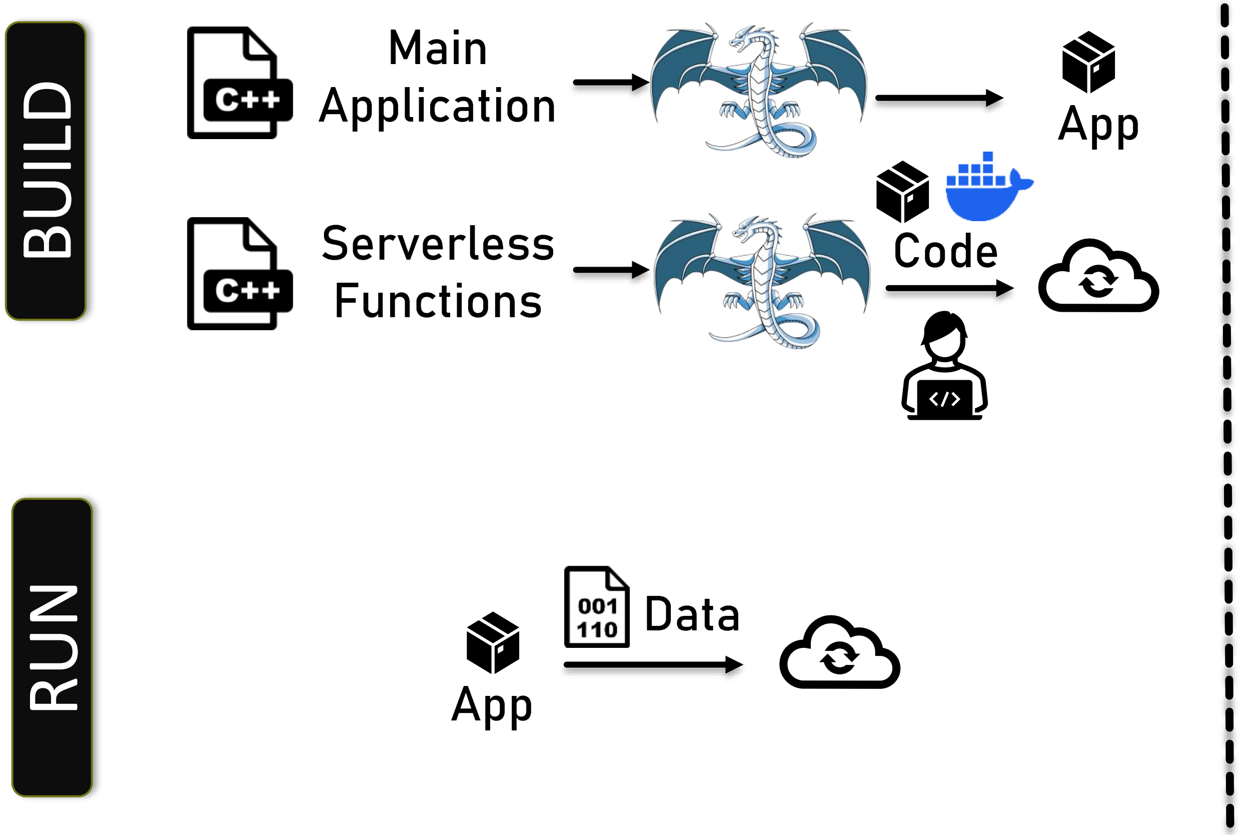


Define and configure generic functions.

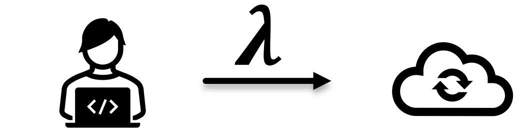


Separate Codebase vs Single-Source Serverless

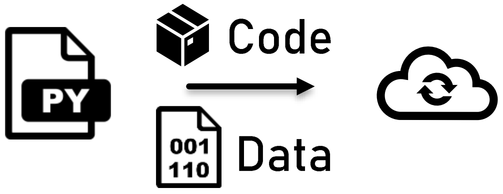
Standard Serverless Workflow



Dynamic Dispatch (Python, Java, Bash)



Define and configure generic functions.



✖ Manual code separation, error-prone

✖ Runtime overhead, not feasible in C/C++

Let’s deploy C++ on AWS Lambda!

```cpp
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{



}
```

Let’s deploy C++ on AWS Lambda!

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }







}
```

Let’s deploy C++ on AWS Lambda!

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std:::to_string(result);



}
```

Let's deploy C++ on AWS Lambda!

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std:::to_string(result);
  return invocation_response:::success(
    response, "application/json"
  );
}
```

Let’s deploy C++ on AWS Lambda!

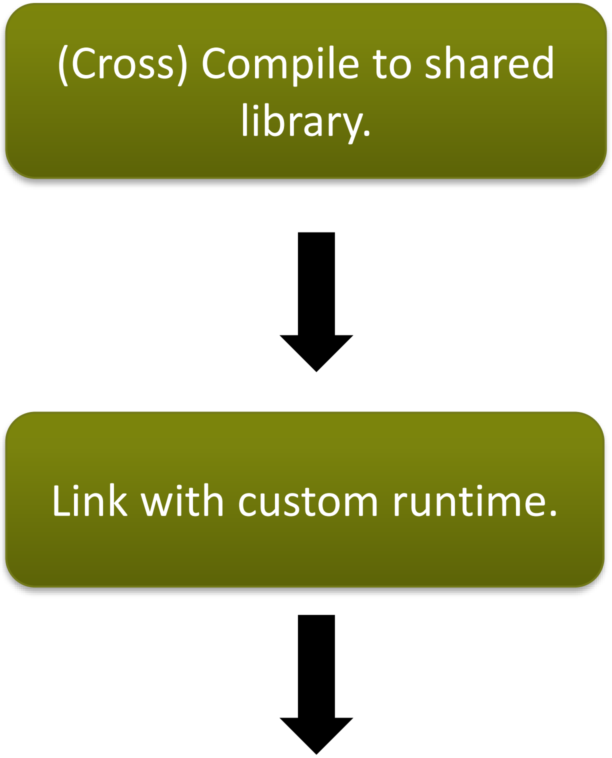
```cpp
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std::to_string(result);
  return invocation_response::success(
    response, "application/json"
  );
}
```

(Cross) Compile to shared library.

↓

Let’s deploy C++ on AWS Lambda!

```cpp
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std:::to_string(result);
  return invocation_response:::success(
    response, "application/json"
  );
}
```

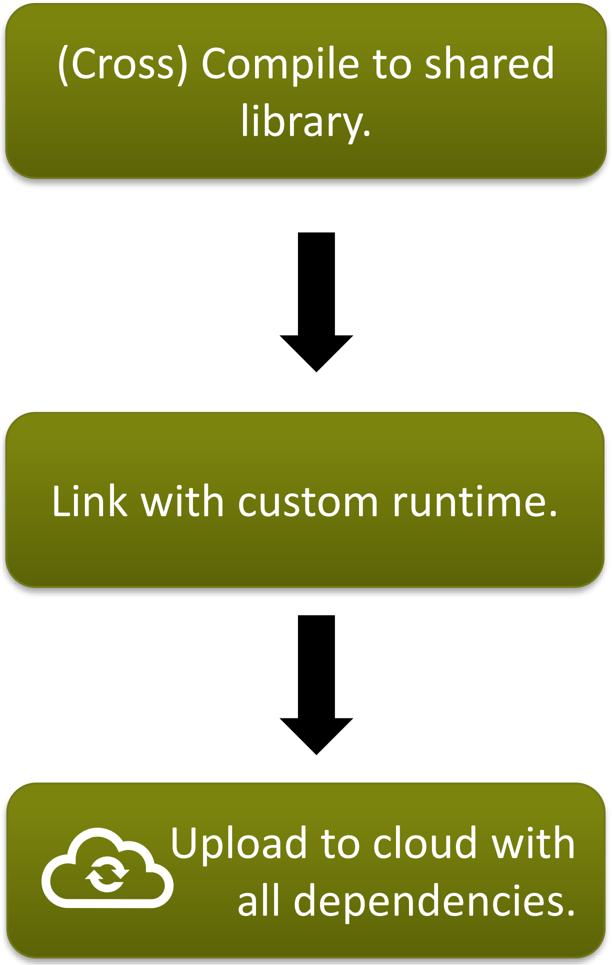


(Cross) Compile to shared library.

Link with custom runtime.

Let’s deploy C++ on AWS Lambda!

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std:::to_string(result);
  return invocation_response:::success(
    response, "application/json"
  );
}
```



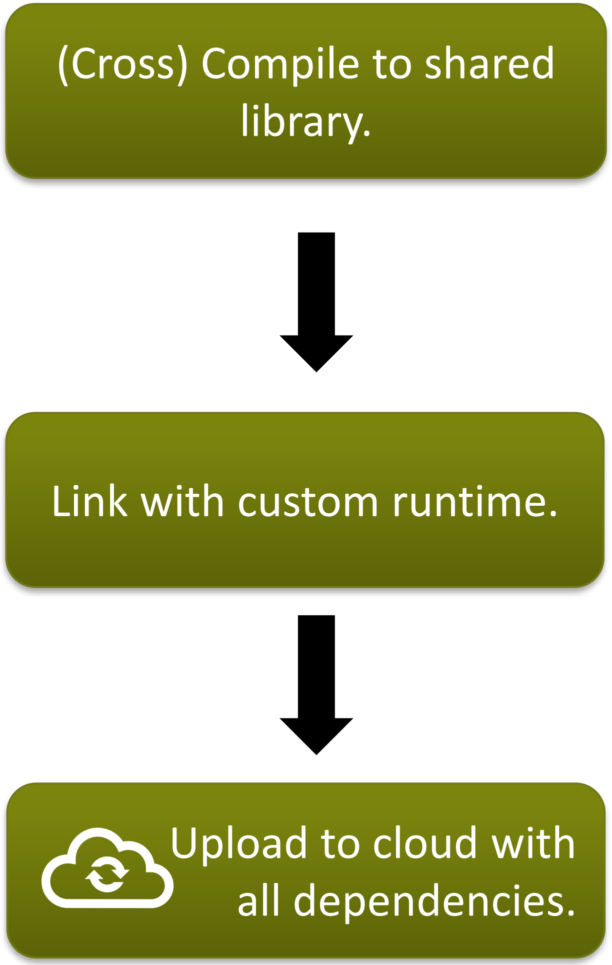
(Cross) Compile to shared library.

Link with custom runtime.

Upload to cloud with all dependencies.

Let’s deploy C++ on AWS Lambda!

```
#include <aws/lambda-runtime/runtime.h>
#include <aws/core/utils/json/JsonSerializer.h>
#include <aws/core/utils/memory/stl/SimpleStringStream.h>

using namespace aws::lambda_runtime;

invocation_response my_handler(invocation_request const& request)
{
  using namespace Aws::Utils::Json;
  JsonValue json(request.payload);
  if (!json.WasParseSuccessful()) {
    return invocation_response::failure(
      "Failed to parse input JSON", "InvalidJSON"
    );
  }
  auto iterations = json.GetInt64("iterations");
  auto result = pi_estimation(iterations);
  auto response = std::to_string(result);
  return invocation_response::success(
    response, "application/json"
  );
}
```



(Cross) Compile to shared library.

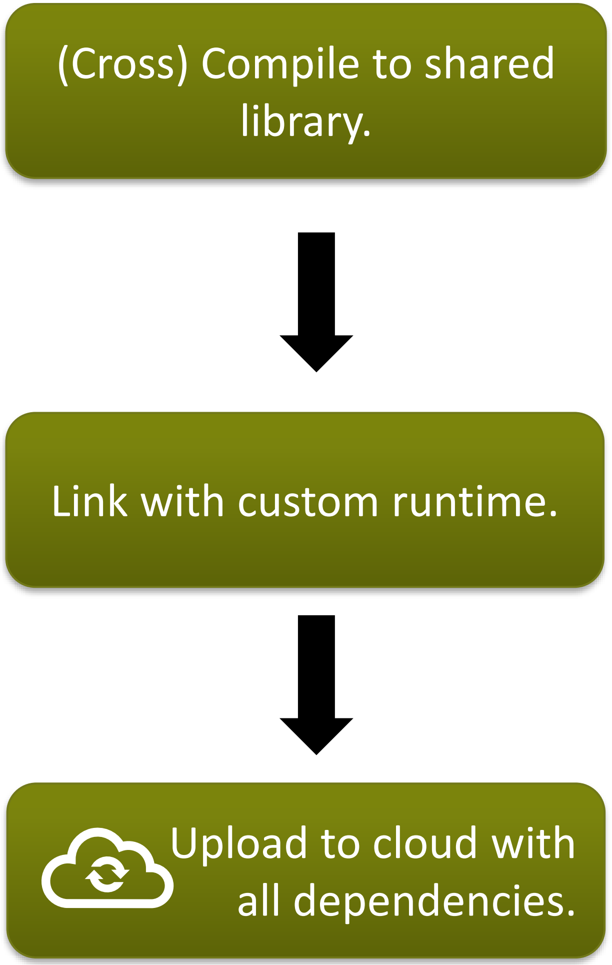
Link with custom runtime.

Upload to cloud with all dependencies.

Let’s deploy C++ on AWS Lambda!

```
#i
#i
#i

us

in                                                           st)
{

}
```

```
bool InvokeFunction(
  const Aws::String& functionName,
  std::shared_ptr<Aws::Lambda::LambdaClient> client,
  int invocations, int &result
)
{
  Aws::Lambda::Model::InvokeRequest invokeRequest;
  invokeRequest.SetFunctionName(functionName);
  invokeRequest.SetInvocationType(
  Aws:::Lambda::Model::InvocationType::RequestResponse);

  std::shared_ptr<Aws::IOStream> payload
        = Aws::MakeShared<Aws::StringStream>();
  Aws::Utils::Json::JsonValue jsonPayload;
  jsonPayload.WithInt64("iterations", iterations);
  *payload << jsonPayload.View().WriteReadable();
  invokeRequest.SetBody(payload);
  invokeRequest.SetContentType("application/json");

  …
}
```



(Cross) Compile to shared library.

Link with custom runtime.

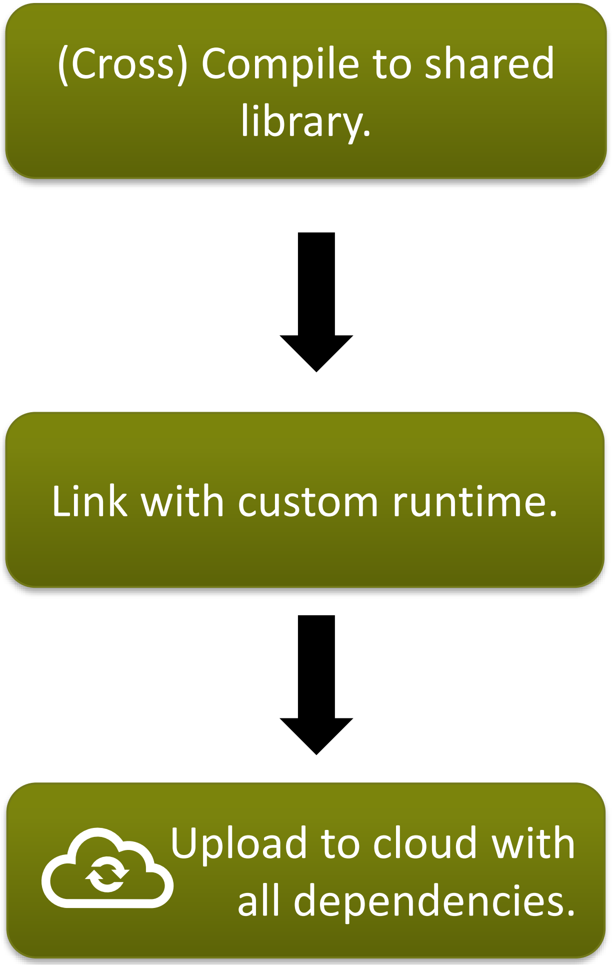
Upload to cloud with all dependencies.

Let’s deploy C++ on AWS Lambda!

```
#i
#i
#i

us

in                                                                   st)
{
                                                                 e);


}
```

```
bool InvokeFunction(

)
{
  …

}
```

```
{
  …

  auto outcome = client->Invoke(invokeRequest);

  if (outcome.IsSuccess()) {
    auto &result = outcome.GetResult();
    Aws::IOStream &payload = result.GetPayload();
    Aws::String functionResult;
    std::getline(payload, functionResult);
    result = std::stoi(functionResult);
    return true;
  } else {
    return false;
  }

}
```



(Cross) Compile to shared library.

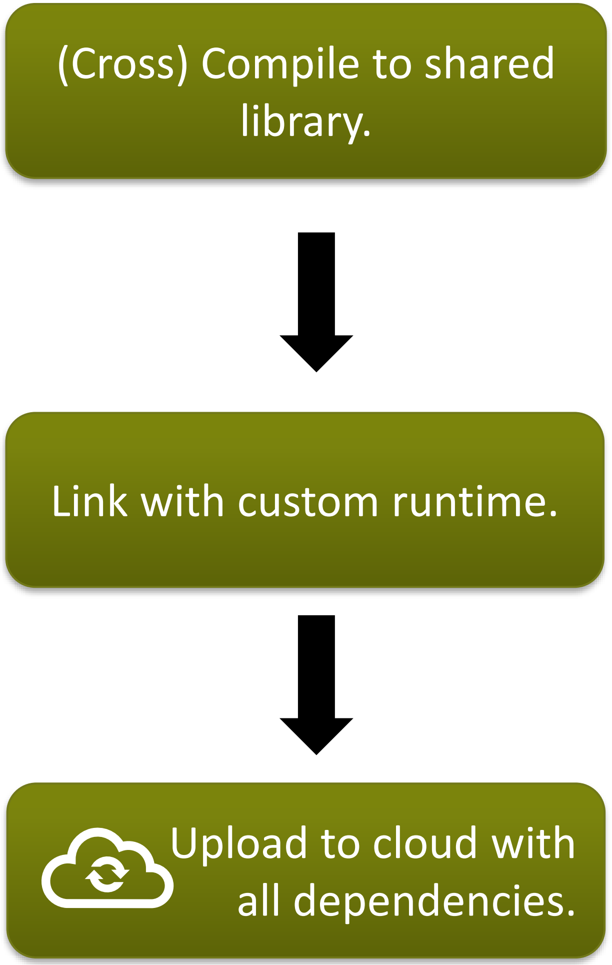
Link with custom runtime.

Upload to cloud with all dependencies.

Let’s deploy C++ on AWS Lambda!

```
#
#
#

u

i                                                                  t)
{



}
```

```
Aws::SDKOptions options;
Aws::InitAPI(options);
Aws::Client::ClientConfiguration clientConfig;
auto m_client = Aws::MakeShared<Aws::Lambda::LambdaClient>(
  ALLOCATION_TAG,
  clientConfig
);

int n = 10000;
int np = 10;
std::vector<int> results(np);
std::vector<std::thread> threads;
for (int i = 0; i < np; i+++) {
  threads.emplace_back([&, i]() {
    InvokeFunction("pi-mc-worker", n / np, results[i]);
  });
}

for (auto &thread : threads) {
  thread.join();
}
auto pi = std:::reduce(results.begin(), results.end()) / np;
```



(Cross) Compile to shared library.

Link with custom runtime.

Upload to cloud with all dependencies.

Challenges of C++ in serverless

Challenges of C++ in serverless

Complex, multi-source project setup

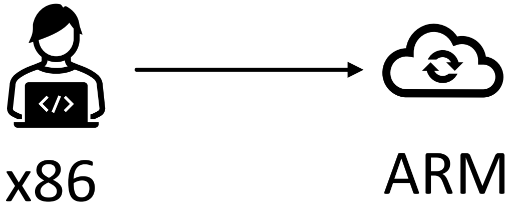
```
#pragma omp but in serverless
for(int i = 0; i < n; ++i)
  pi_mc(i);
```

Challenges of C++ in serverless

Complex, multi-source project setup

```
#pragma omp but in serverless
for(int i = 0; i < n; ++i)
  pi_mc(i);
```

Cross-compiled environments

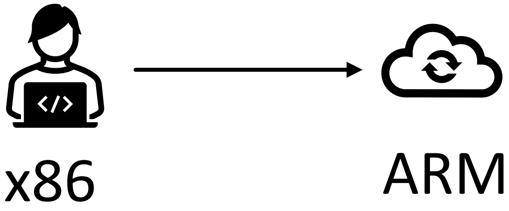


Challenges of C++ in serverless

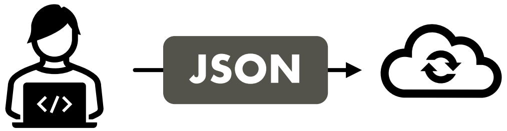
Complex, multi-source project setup

```
#pragma omp but in serverless
for(int i = 0; i < n; ++i)
  pi_mc(i);
```

Cross-compiled environments



Lack of static typing



Manual verification?
JSON Schema?

Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless::dispatch(aws, fn, result);
  cppless::wait(aws, np);

  auto pi = std::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless::dispatch(aws, fn, result);
  cppless::wait(aws, np);

  auto pi = std::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

- **C++ abstraction for cloud provider APIs.**

  Avoids the vendor lock-in and simplifies invocations.

Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless::dispatch(aws, fn, result);
  cppless::wait(aws, np);

  auto pi = std::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.

Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.

Automatically compiled to a cloud function.

Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless::dispatch(aws, fn, result);
  cppless::wait(aws, np);

  auto pi = std::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

⬢ **C++ abstraction for cloud provider APIs.**

Avoids the vendor lock-in and simplifies invocations.

⬢ **Serverless function as C++ lambda expression.**

Automatically compiled to a cloud function.

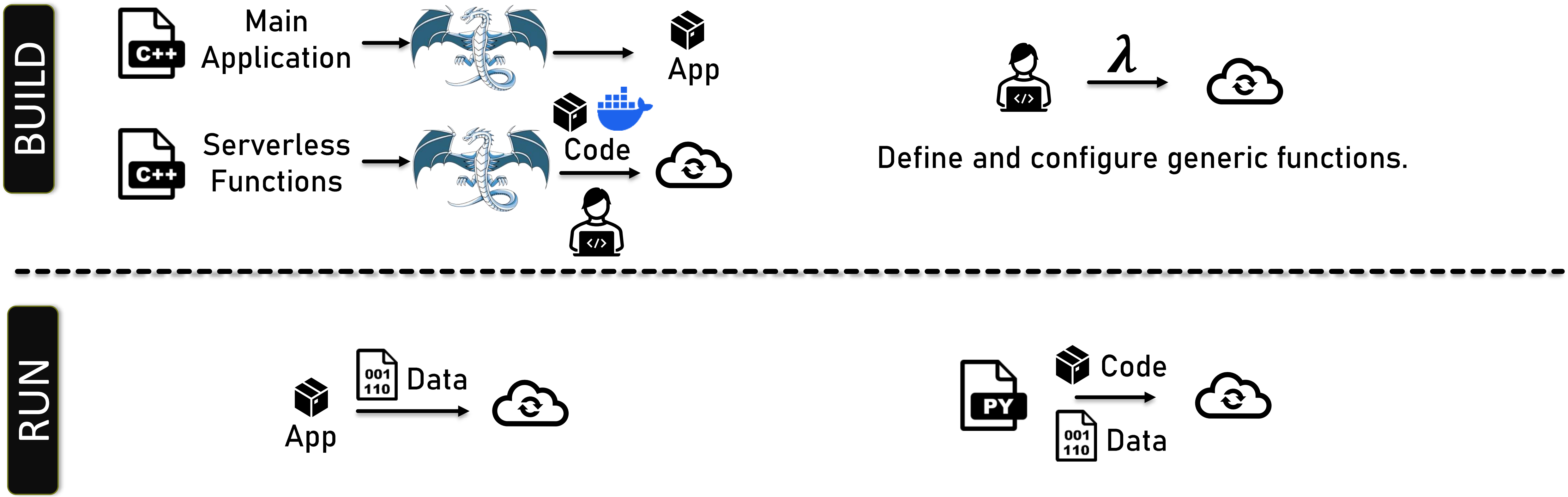
⬢ **Integrated invocation of the function.**

Automatic serialization and type checking.

Cppless: How it Works?

Standard Serverless Workflow

Dynamic Dispatch (Python, Java, Bash)



Define and configure generic functions.

✗ Manual code separation, error-prone

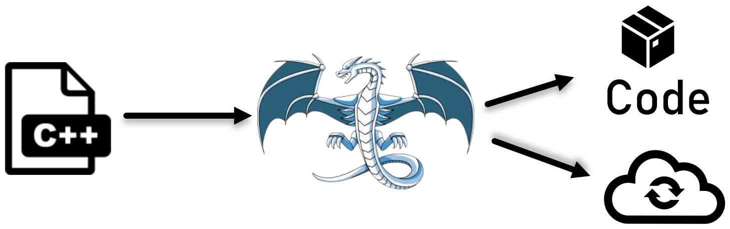
✗ Runtime overhead, not feasible in C/C++

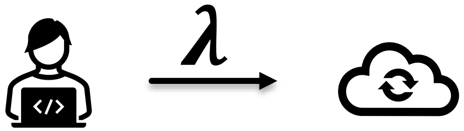
Cppless: How it Works?

Cppless: Single-Source Serverless

Dynamic Dispatch (Python, Java, Bash)

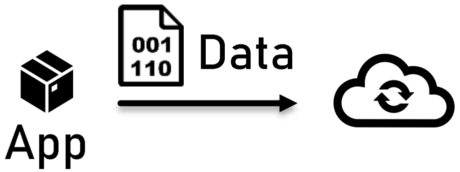
BUILD

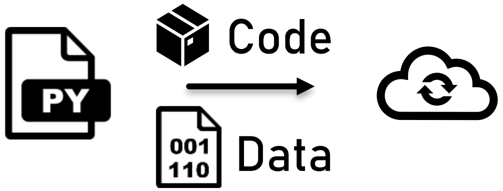




Define and configure generic functions.

RUN





✗ Manual code separation, error-prone

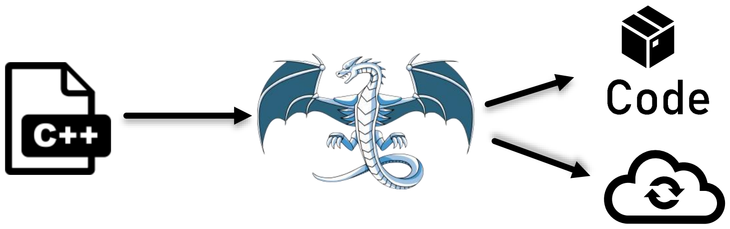
✗ Runtime overhead, not feasible in C/C++

Cppless: How it Works?

Cppless: Single-Source Serverless

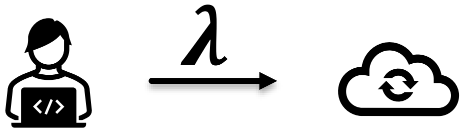
Dynamic Dispatch (Python, Java, Bash)

BUILD



C++

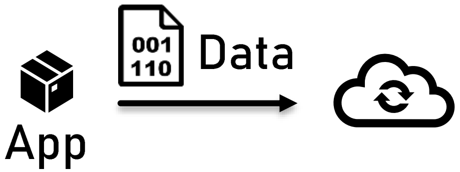
Code



λ

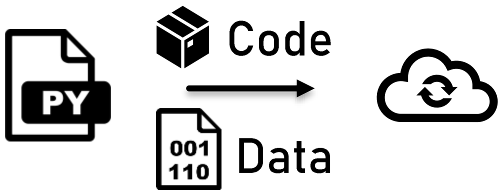
Define and configure generic functions.

RUN



App

001 110 Data



PY

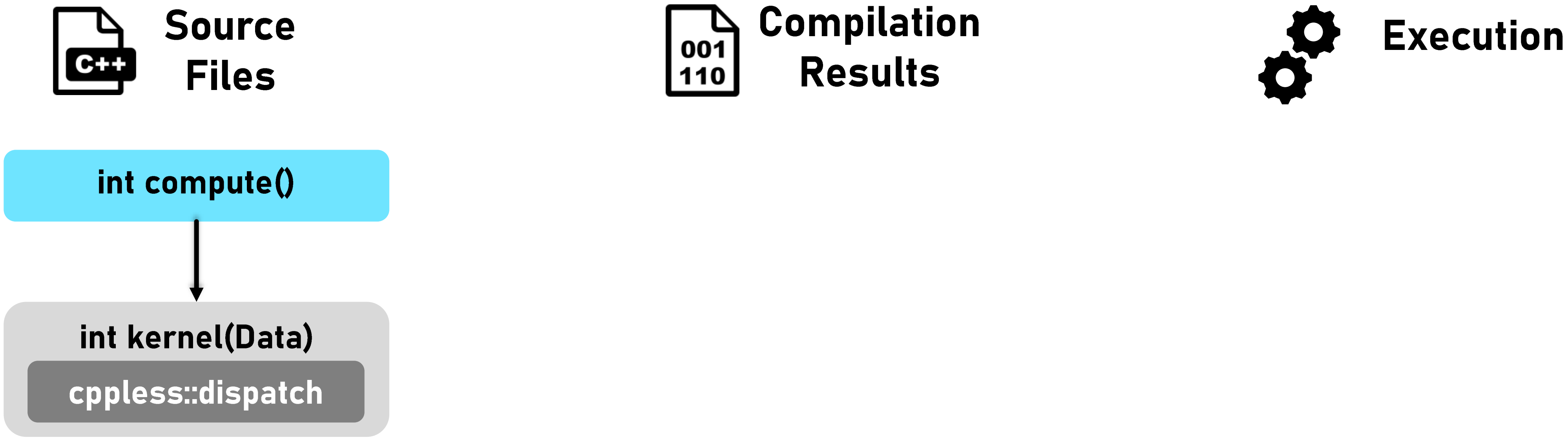
Code

001 110 Data

✓ Automatic code deployment
Integrated data serialization

✗ Runtime overhead, not feasible in C/C++

From C++ Lambda to Serverless Lambda



Source Files

Compilation Results

Execution

int compute()

int kernel(Data)

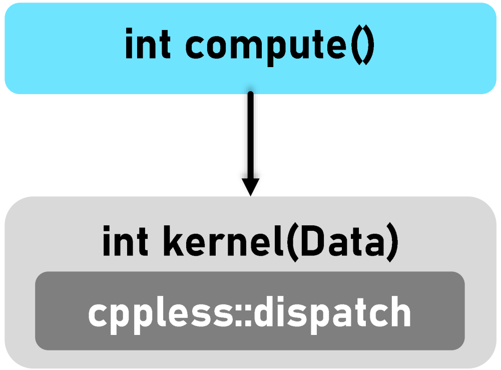
cppless::dispatch

From C++ Lambda to Serverless Lambda

Source Files

Compilation Results

Execution



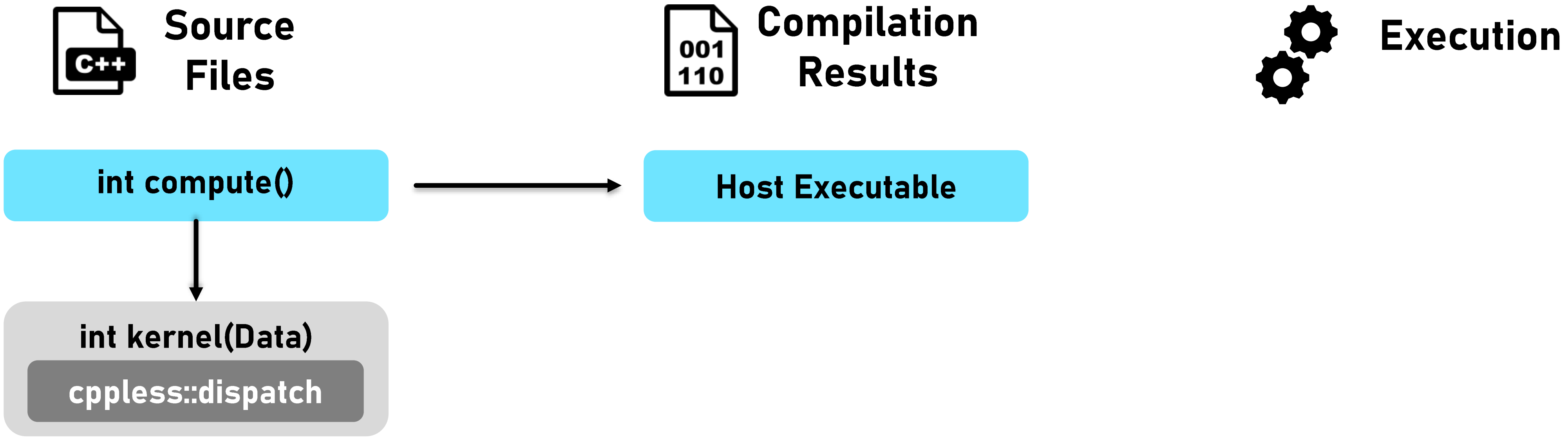
int compute()

int kernel(Data)

cppless::dispatch

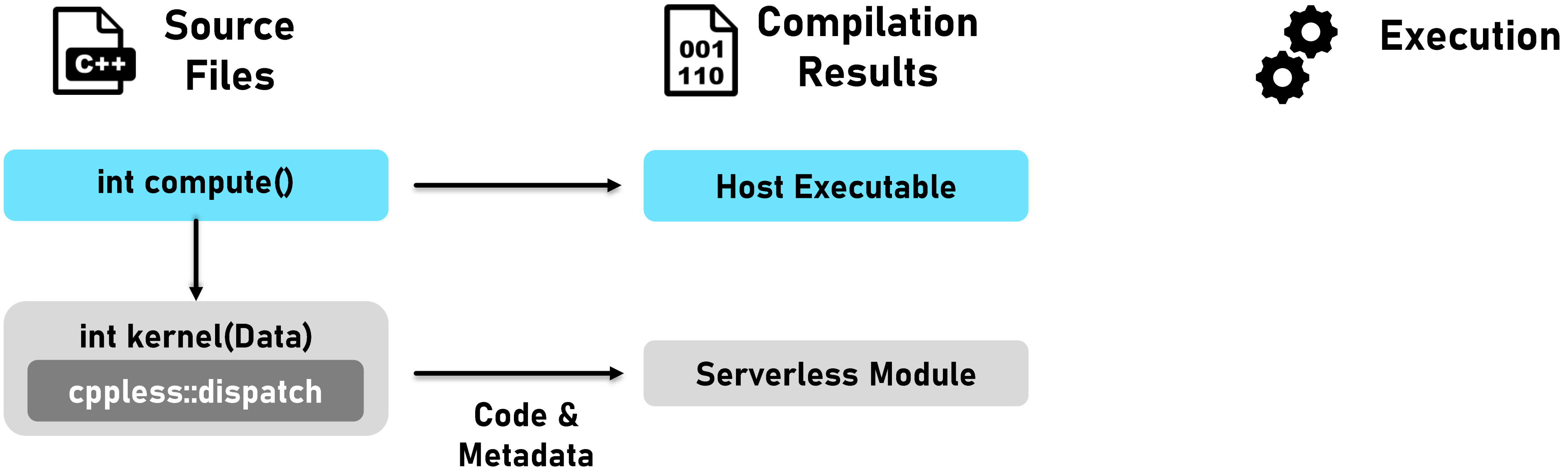
Alternative Entry Points

From C++ Lambda to Serverless Lambda



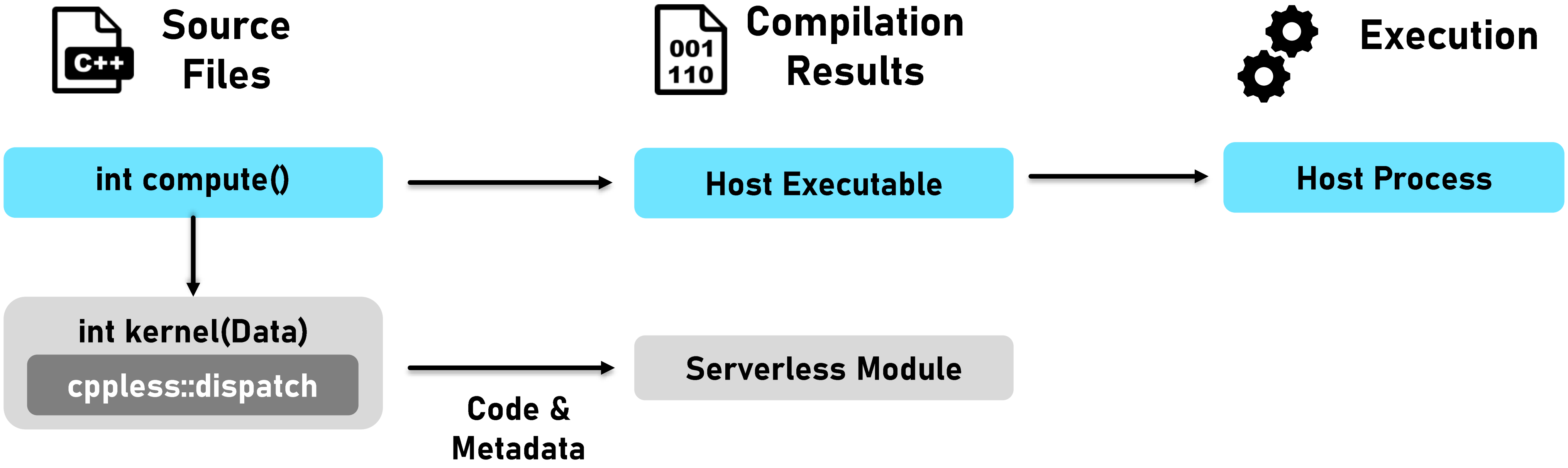
Alternative Entry Points

From C++ Lambda to Serverless Lambda



Alternative Entry Points

From C++ Lambda to Serverless Lambda



Source Files

Compilation Results

Execution

int compute()

Host Executable

Host Process

int kernel(Data)

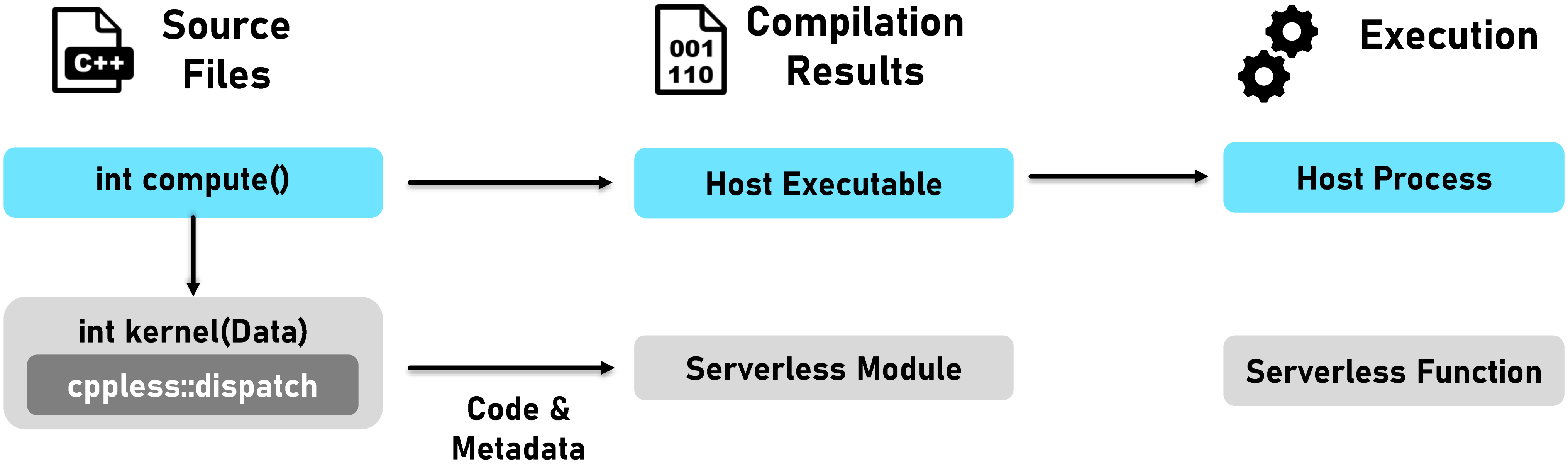
cppless::dispatch

Code & Metadata

Serverless Module

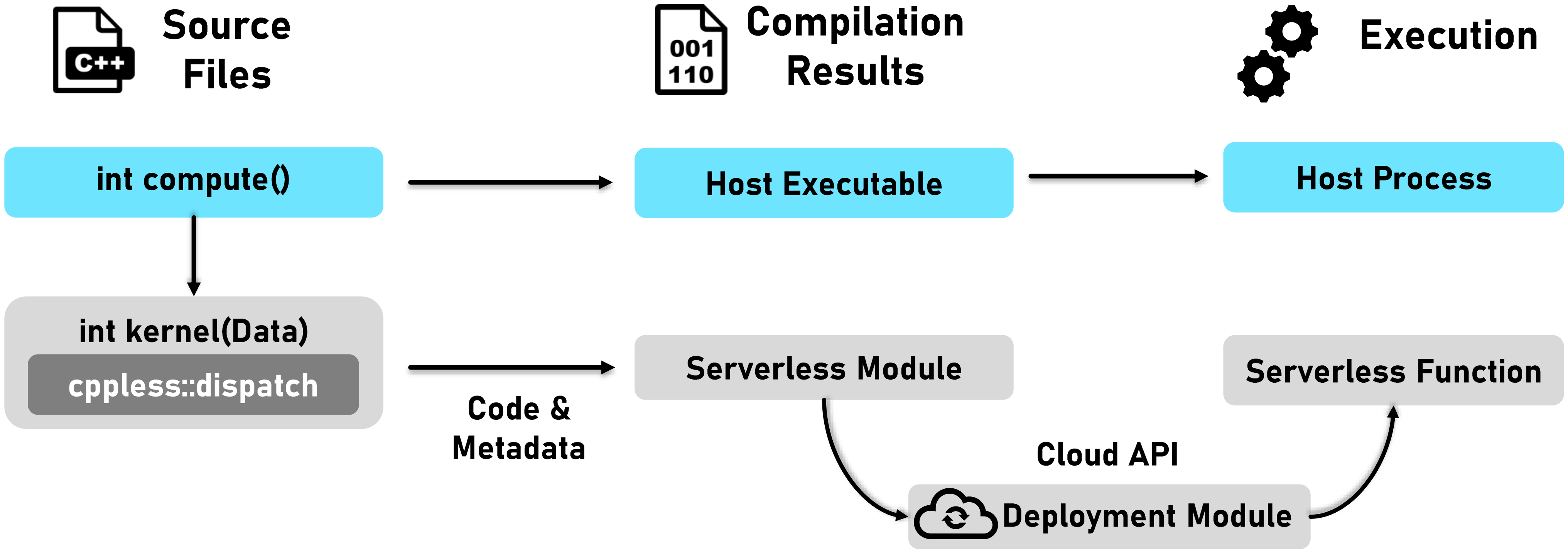
Alternative Entry Points

From C++ Lambda to Serverless Lambda



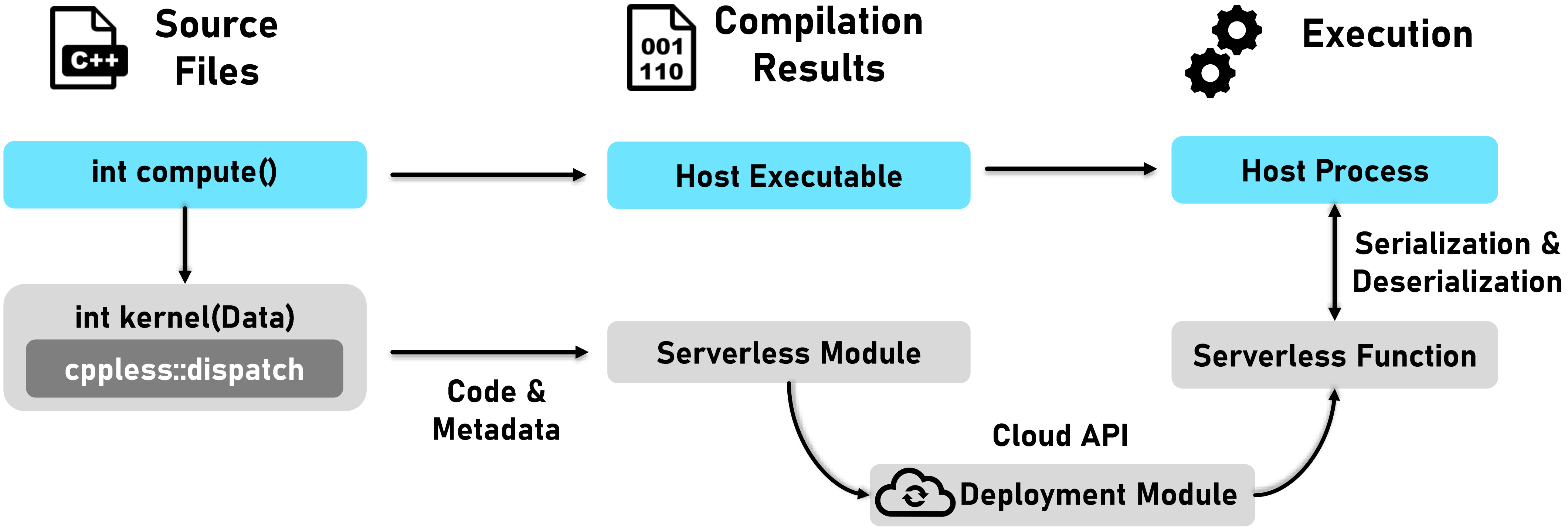
Alternative Entry Points

From C++ Lambda to Serverless Lambda



Alternative Entry Points

From C++ Lambda to Serverless Lambda

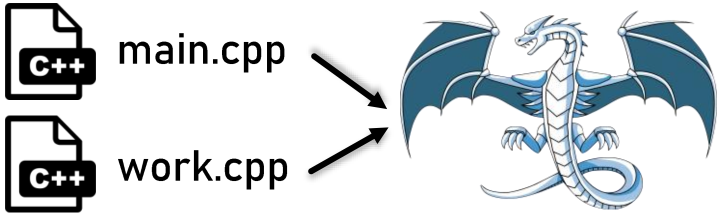


Alternative Entry Points

λ Lambda Reflection

From C++ Lambda to Serverless Lambda: Alternative Entry Points

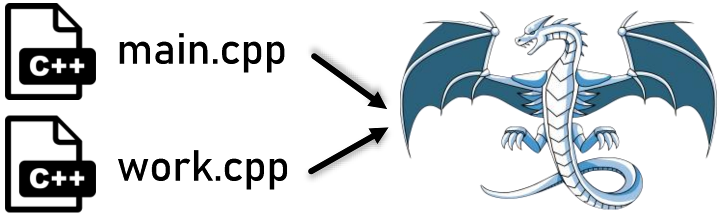
From C++ Lambda to Serverless Lambda: Alternative Entry Points



main.cpp

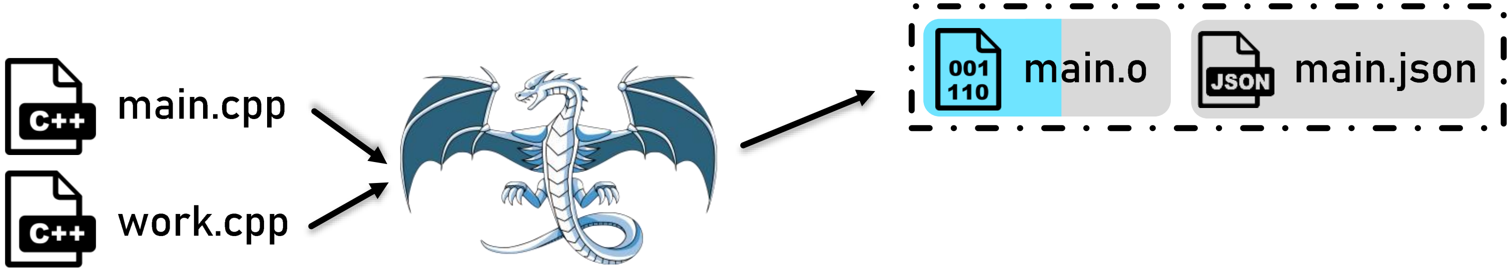
work.cpp

From C++ Lambda to Serverless Lambda: Alternative Entry Points



clang –cppless -falt-entry

From C++ Lambda to Serverless Lambda: Alternative Entry Points



main.cpp

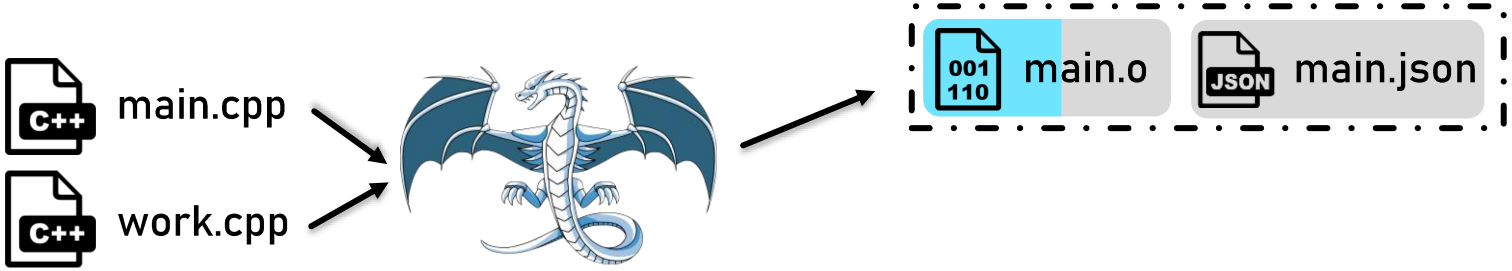
work.cpp

main.o

main.json

clang –cppless -falt-entry

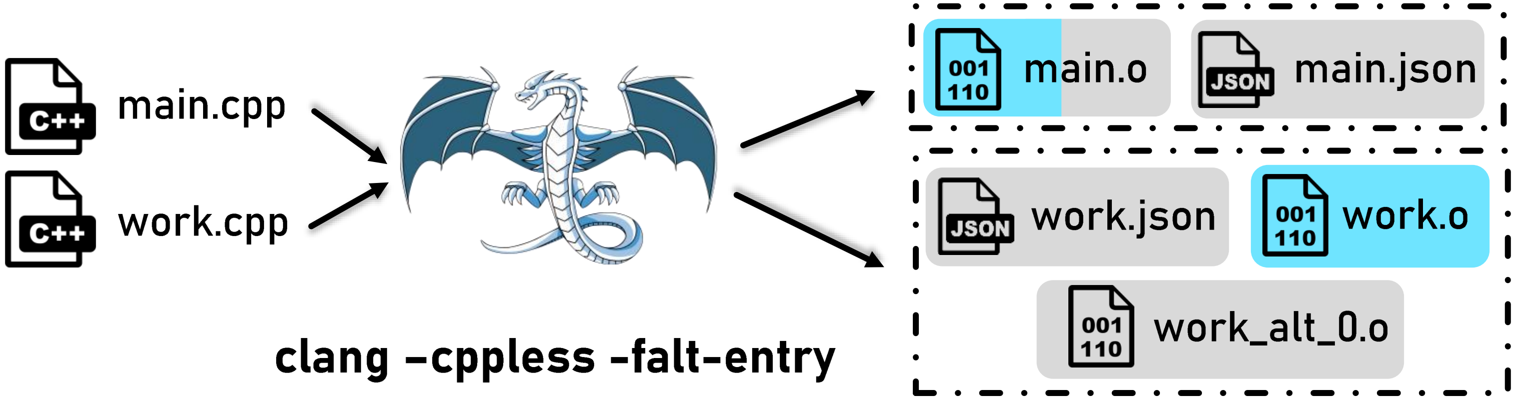
From C++ Lambda to Serverless Lambda: Alternative Entry Points



clang –cppless -falt-entry

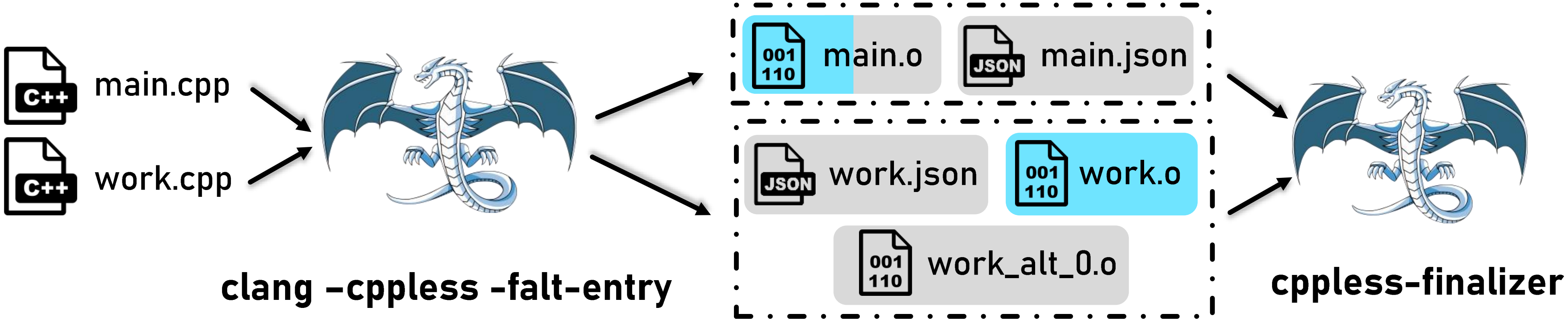
```
"entry_points": [{
    "original_function_name": "mangled_C++_function_name", "filename": "dispatcher_aws_alt_0",
    "user_meta": {
      "ephemeral_storage": 512, "memory": 1024, "timeout": 10,
      "identifier": "./examples/aws/dispatcher.cpp@..."
    }
}]
```

From C++ Lambda to Serverless Lambda: Alternative Entry Points



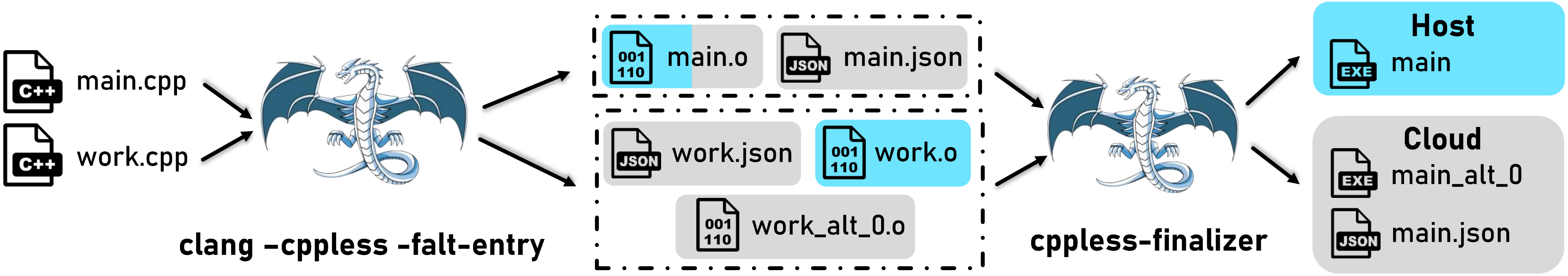
```
"entry_points": [{
    "original_function_name": "mangled_C++_function_name", "filename": "dispatcher_aws_alt_0",
    "user_meta": {
      "ephemeral_storage": 512, "memory": 1024, "timeout": 10,
      "identifier": "./examples/aws/dispatcher.cpp@..."
    }
}]
```

From C++ Lambda to Serverless Lambda: Alternative Entry Points



```
"entry_points": [{
    "original_function_name": "mangled_C++_function_name", "filename": "dispatcher_aws_alt_0",
    "user_meta": {
      "ephemeral_storage": 512, "memory": 1024, "timeout": 10,
      "identifier": "./examples/aws/dispatcher.cpp@..."
    }
}]
```

From C++ Lambda to Serverless Lambda: Alternative Entry Points



main.cpp

work.cpp

clang –cppless -falt-entry

main.o

main.json

work.json

work.o

work_alt_0.o

cppless-finalizer

Host

main

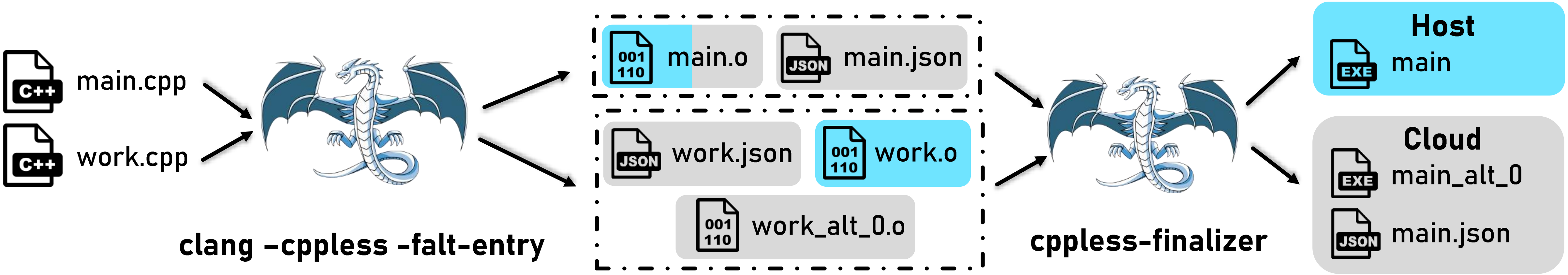
Cloud

main_alt_0

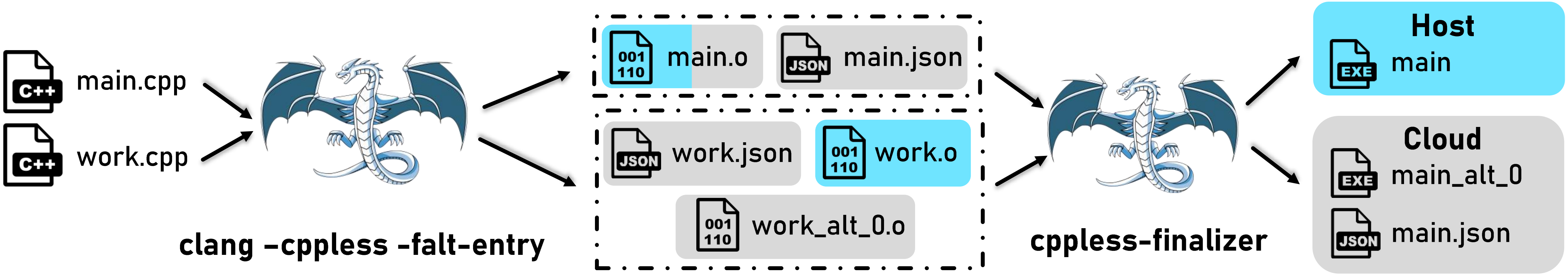
main.json

```
"entry_points": [{
    "original_function_name": "mangled_C++_function_name", "filename": "dispatcher_aws_alt_0",
    "user_meta": {
      "ephemeral_storage": 512, "memory": 1024, "timeout": 10,
      "identifier": "./examples/aws/dispatcher.cpp@..."
    }
}]
```

From C++ Lambda to Serverless Lambda: Alternative Entry Points



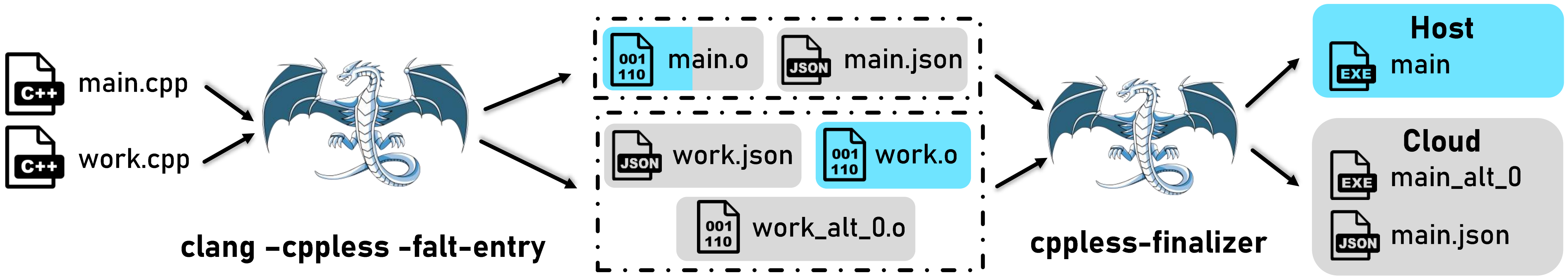
From C++ Lambda to Serverless Lambda: Alternative Entry Points



Clang Frontend

- ✓ Parse alternative entry points
- ✓ Lambda reflection support

From C++ Lambda to Serverless Lambda: Alternative Entry Points



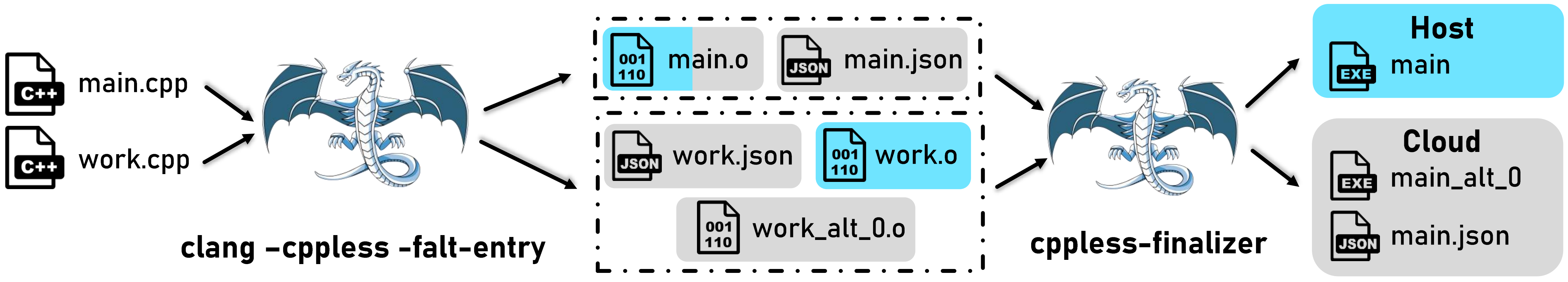
Clang Frontend

- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend

- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

From C++ Lambda to Serverless Lambda: Alternative Entry Points



Clang Frontend

- ✓ Parse alternative entry points
- ✓ Lambda reflection support

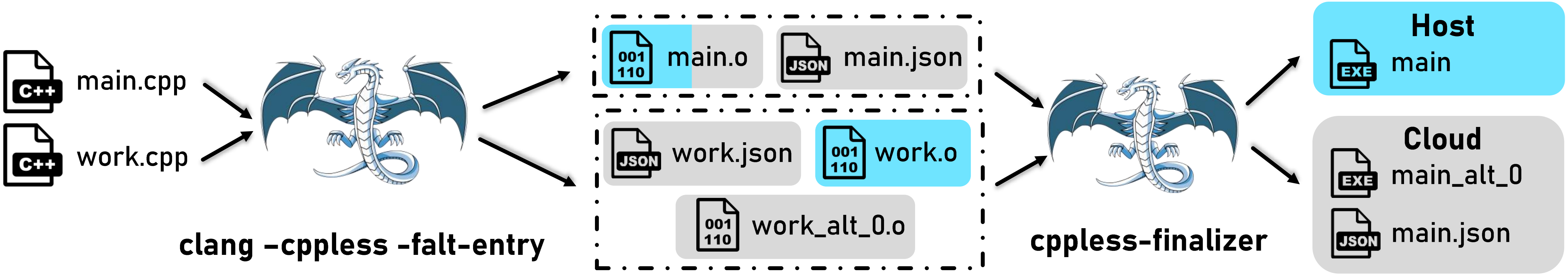
LLVM Backend

- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer

- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

From C++ Lambda to Serverless Lambda: Alternative Entry Points



Clang Frontend

- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend

- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer

- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

Total changes: 963 lines of code added, 192 removed

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```cpp
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

```
42
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

```
42
```

From C++ Lambda to Serverless Lambda: Serialization & Reflection

```
template<typename Func>
void serialize_lambda(Func && lambda) {

  double arg1 = 42; std::string arg2 = "capture";
  auto lambda = [arg1, arg2]() { std::cout << arg1 << " " << arg2 << std::endl; };

  constexpr int capture_count = decltype(lambda)::capture_count();

  auto captured_arg1 = lambda.capture<0>();
  std::cout << captured_arg1 << std::endl;
  lambda.capture<0>() = 43;

  lambda();
}
```

```
42

43 capture
```

From C++ Lambda to Serverless Lambda: Dispatch

```
template<typename Func>
void dispatch_function(Func && lambda) {

  auto first_capture = serialize(lambda.capture<0>());

  auto func_id = __builtin_unique_stable_name(decltype(lambda));

  invoke(func_id, first_capture);
}
```

From C++ Lambda to Serverless Lambda: Dispatch

```
template<typename Func>
void dispatch_function(Func && lambda) {

  auto first_capture = serialize(lambda.capture<0>());

  auto func_id = __builtin_unique_stable_name(decltype(lambda));

  invoke(func_id, first_capture);
}
```

From C++ Lambda to Serverless Lambda: Dispatch

```
template<typename Func>
void dispatch_function(Func && lambda) {

  auto first_capture = serialize(lambda.capture<0>());

  auto func_id = __builtin_unique_stable_name(decltype(lambda));

  invoke(func_id, first_capture);
}
```

SYCL™

From C++ Lambda to Serverless Lambda: Dispatch

```
template<typename Func>
void dispatch_function(Func && lambda) {

  auto first_capture = serialize(lambda.capture<0>());

  auto func_id = __builtin_unique_stable_name(decltype(lambda));

  invoke(func_id, first_capture);
}
```

Internally uses **__attribute((entry))**

From C++ Lambda to Serverless Lambda: Dispatch

```
template<typename Func>
void dispatch_function(Func && lambda) {

  auto first_capture = serialize(lambda.capture<0>());

  auto func_id = __builtin_unique_stable_name(decltype(lambda));

  invoke(func_id, first_capture);
}
```

Internally uses **__attribute((entry))**

nghttp2.org

HTTP/2 C library and tools

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

Create C++ executable.

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

Link Cppless runtime.

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

Enable Cppless, link Lambda C++ runtime.

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

Create nested Cmake project.

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

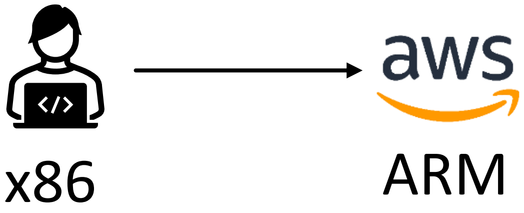
Create nested Cmake project.

Deploy code package.

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

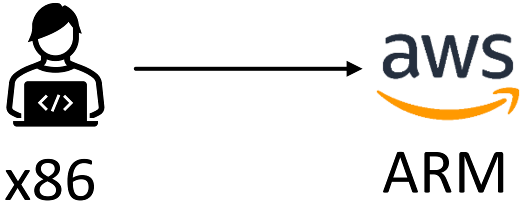
```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```



From C++ Lambda to Serverless Lambda: Build System Integration

CMake

```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```



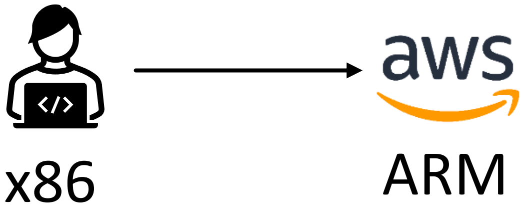
AWS Lambda: x86 vs Graviton CPU performance

From C++ Lambda to Serverless Lambda: Build System Integration

CMake

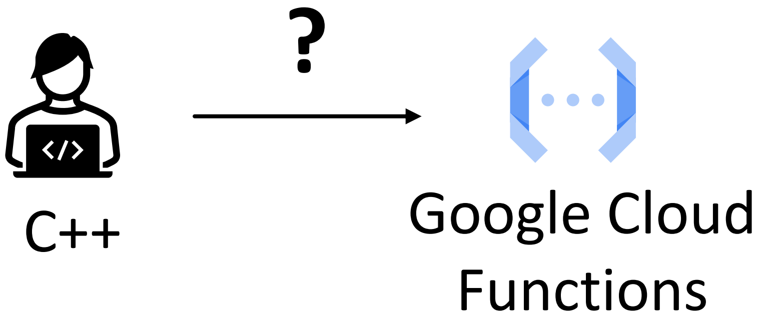
```
add_executable(parallel_pi parallel_pi.cpp)

target_link_libraries(parallel_pi PRIVATE cppless::cppless)

aws_lambda_target(parallel_pi)

aws_lambda_serverless_target(parallel_pi)
```

AWS Lambda: x86 vs Graviton CPU performance



```
set(COMPILER_FLAGS "--target=aarch64-amazon-linux --sysroot=${SYSROOT} --gcc-toolchain=${SYSROOT}/usr/")
set(CMAKE_C_FLAGS "${COMPILER_FLAGS} ")
set(CMAKE_CXX_FLAGS "${COMPILER_FLAGS} ")
```

From C++ Lambda to Serverless Lambda: WASM

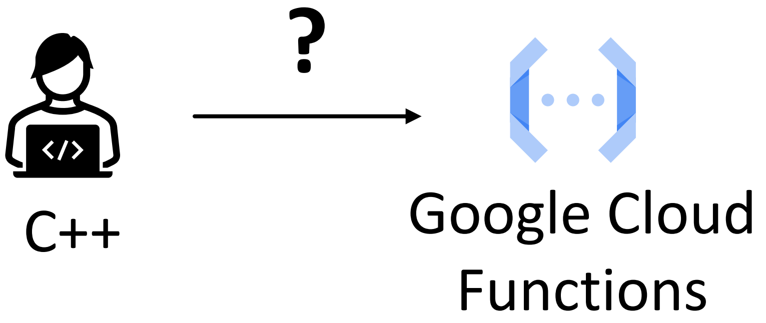


?

C++

Google Cloud Functions

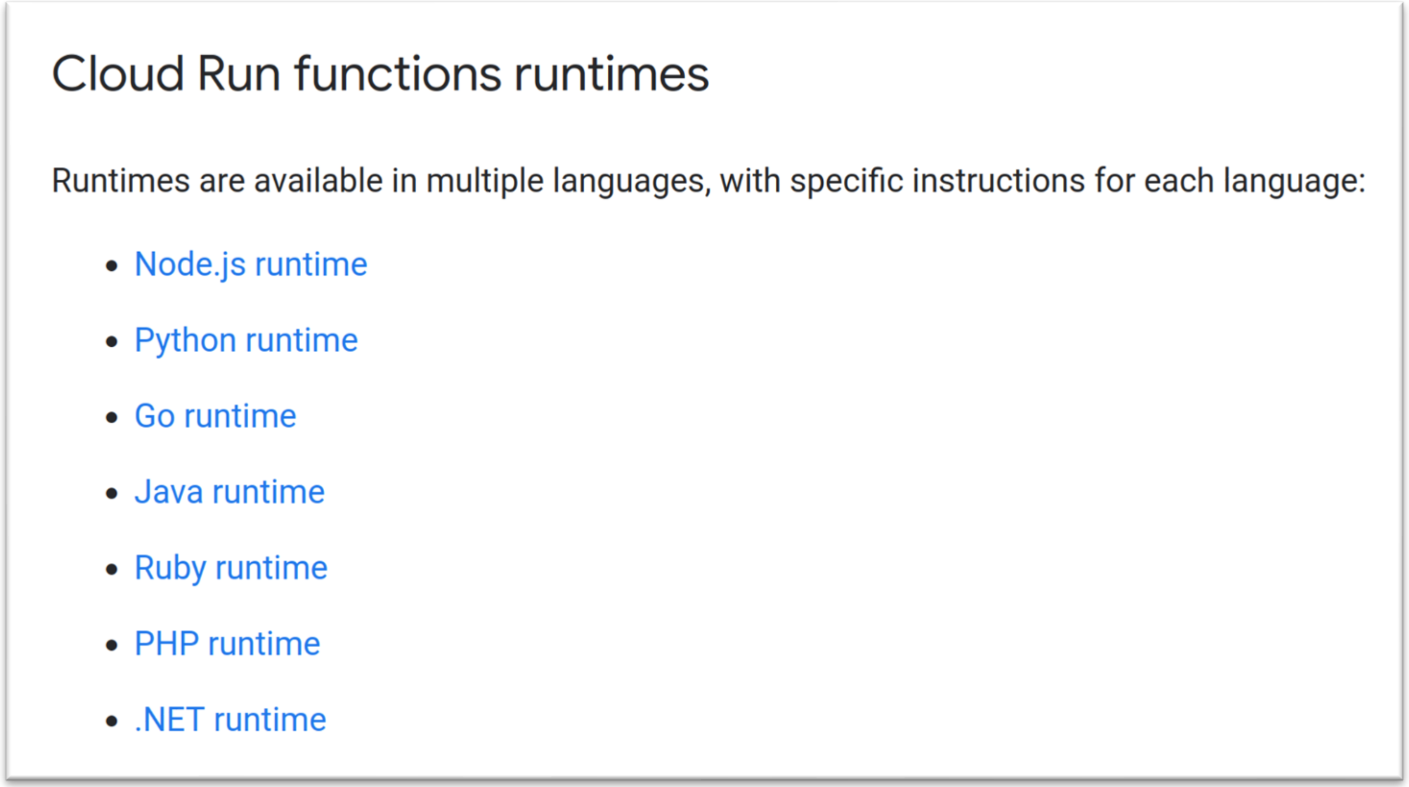
From C++ Lambda to Serverless Lambda: WASM



C++

?

Google Cloud Functions

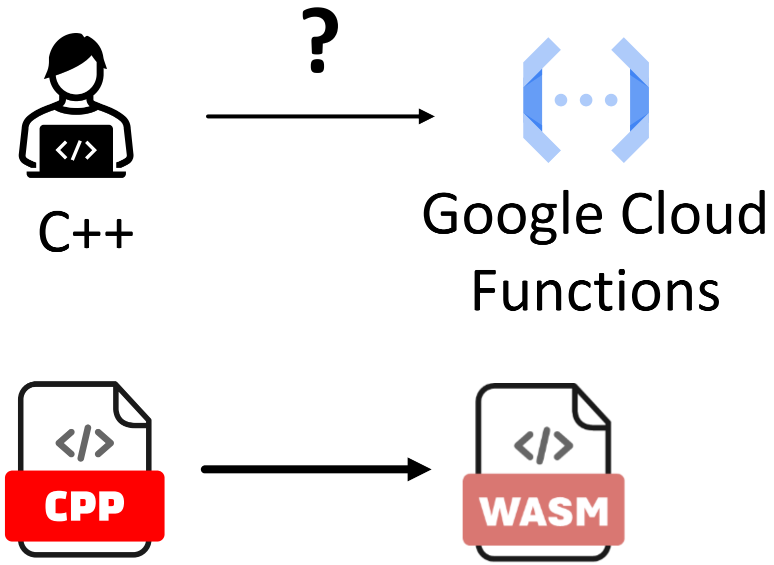


Cloud Run functions runtimes

Runtimes are available in multiple languages, with specific instructions for each language:

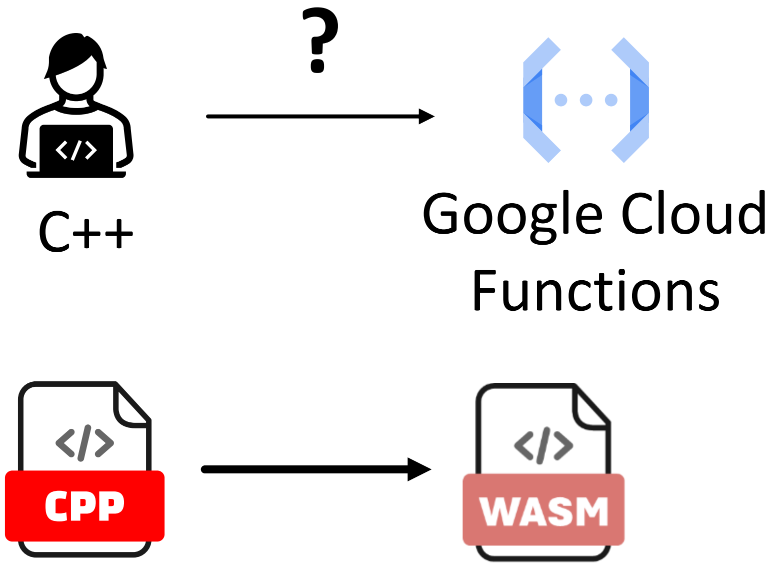
- Node.js runtime
- Python runtime
- Go runtime
- Java runtime
- Ruby runtime
- PHP runtime
- .NET runtime

From C++ Lambda to Serverless Lambda: WASM



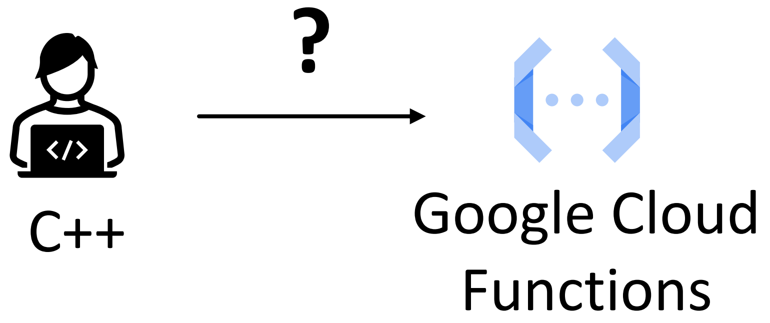
From C++ Lambda to Serverless Lambda: WASM

BUILD



```
clang++ -fno-exceptions -target wasm32-wasi --sysroot=$(pwd)/wasi-sysroot/ -Wl,--no-entry -Wl,--export-all
-mexec-model=reactor -o function/function.wasm function/function.cpp
```

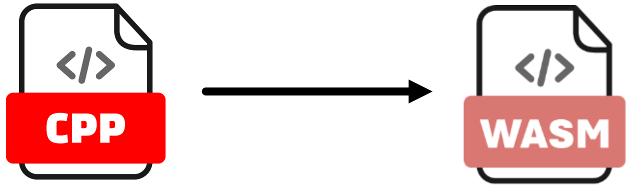
From C++ Lambda to Serverless Lambda: WASM



C++

Google Cloud Functions

BUILD



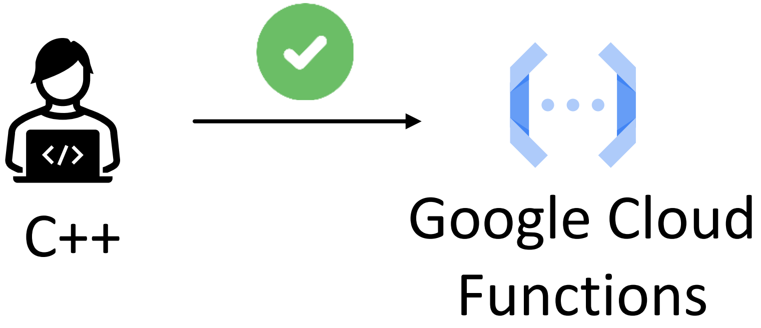
```
clang++ -fno-exceptions -target wasm32-wasi --sysroot=$(pwd)/wasi-sysroot/ -Wl,--no-entry -Wl,--export-all
-mexec-model=reactor -o function/function.wasm function/function.cpp
```

RUN

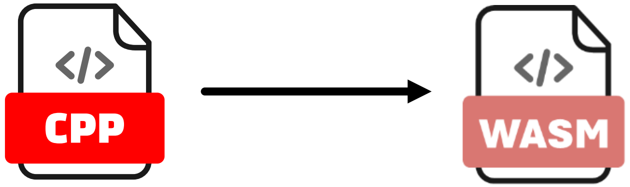
Entrypoint

C++ Function

From C++ Lambda to Serverless Lambda: WASM

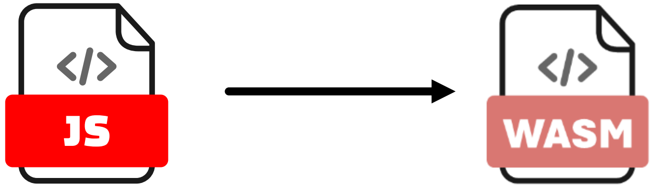


BUILD



```
clang++ -fno-exceptions -target wasm32-wasi --sysroot=$(pwd)/wasi-sysroot/ -Wl,--no-entry -Wl,--export-all
-mexec-model=reactor -o function/function.wasm function/function.cpp
```

RUN

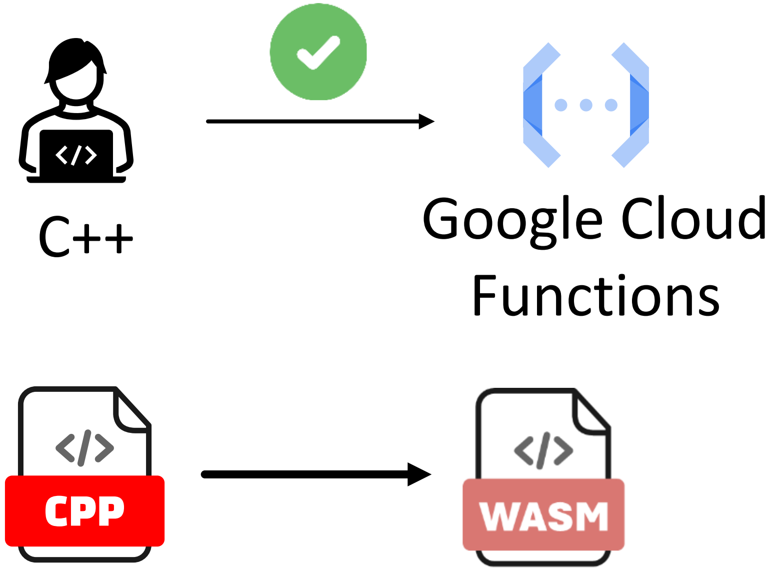


Entrypoint

C++ Function

From C++ Lambda to Serverless Lambda: WASM

BUILD

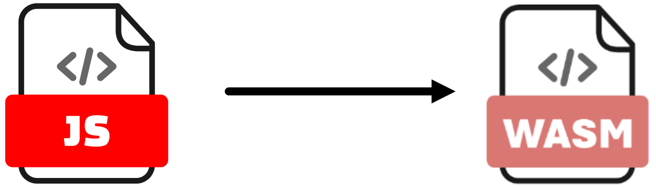


C++

Google Cloud Functions

```
clang++ -fno-exceptions -target wasm32-wasi --sysroot=$(pwd)/wasi-sysroot/ -Wl,--no-entry -Wl,--export-all
-mexec-model=reactor -o function/function.wasm function/function.cpp
```

RUN



Entrypoint

C++ Function

Validated prototype!
Cppless toolchain:
future work.

Evaluation aws

Evaluation aws

1 How fast can we scale up functions?

Evaluation aws

1. How fast can we scale up functions?

2. How Cppless functions perform on tasking problems?

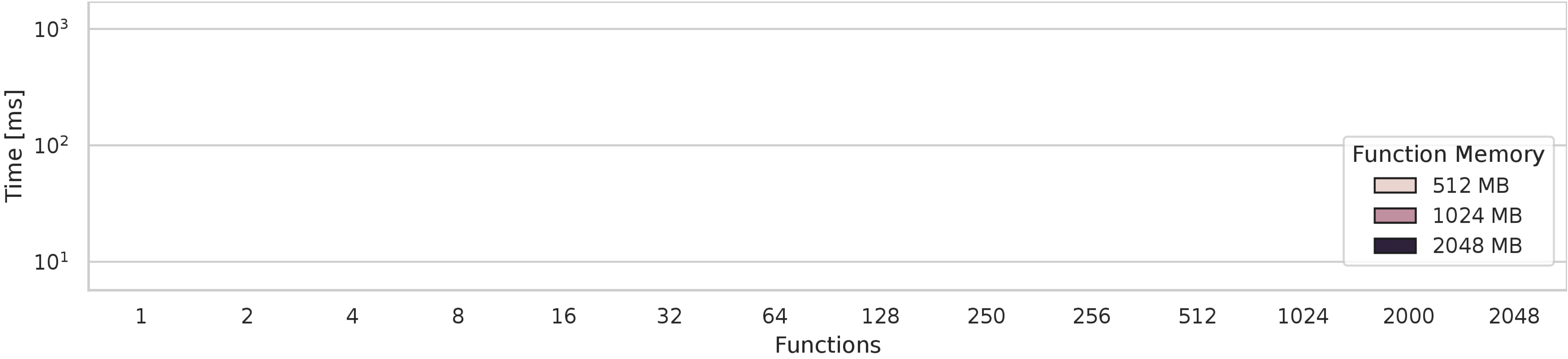
Evaluation aws

1 How fast can we scale up functions?

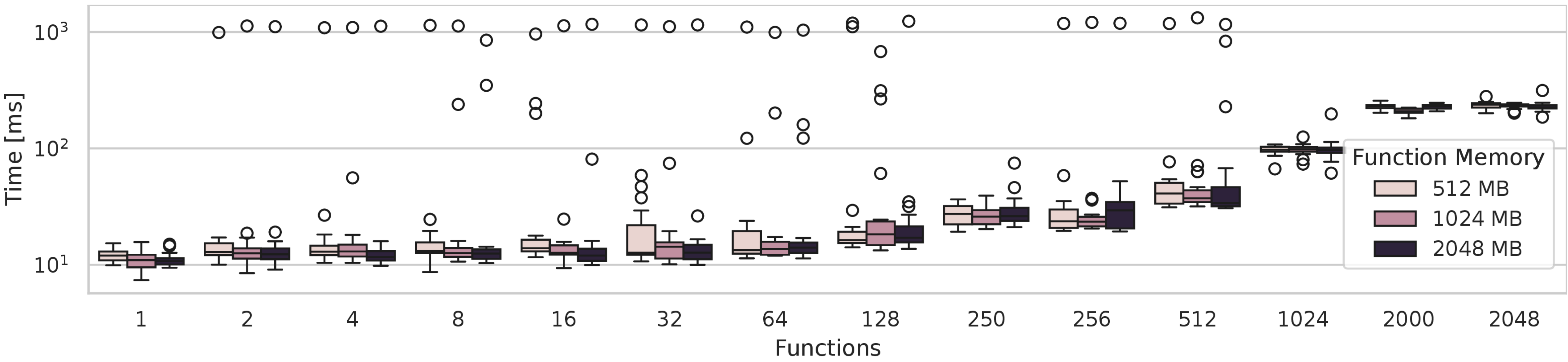
2 How Cppless functions perform on tasking problems?

3 How Cppless functions perform on Monte Carlo problems?

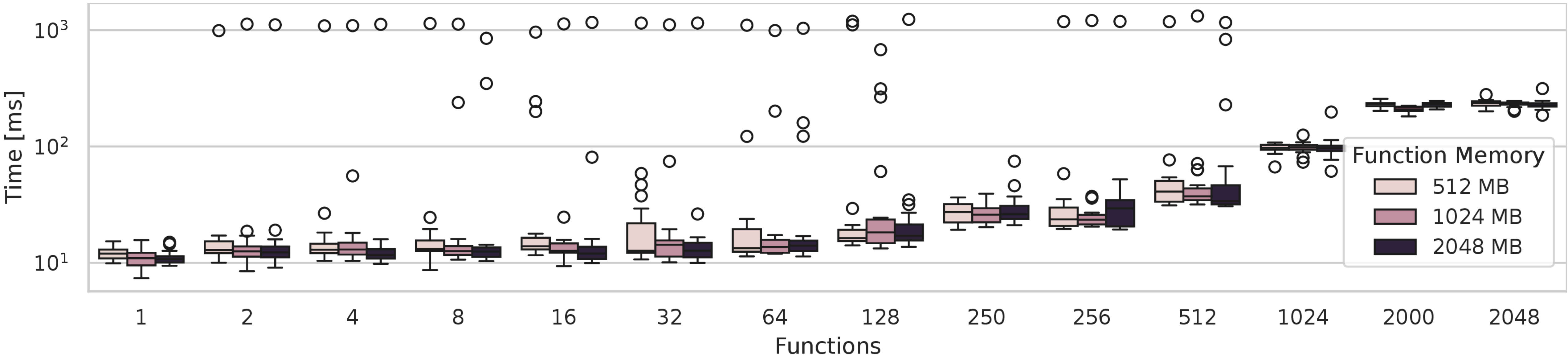
Evaluation: Warm Invocations 1



Evaluation: Warm Invocations 1



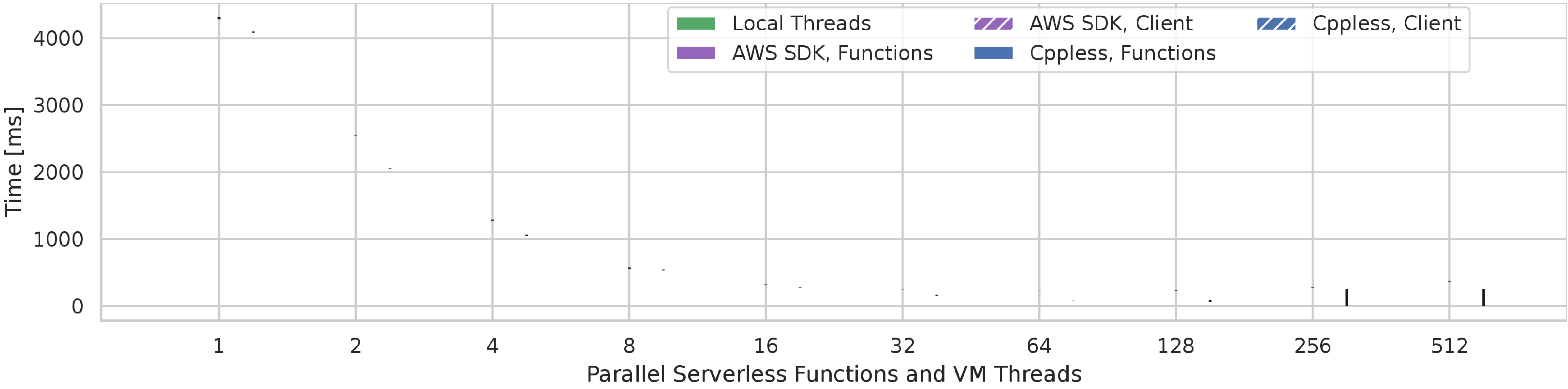
Evaluation: Warm Invocations 1



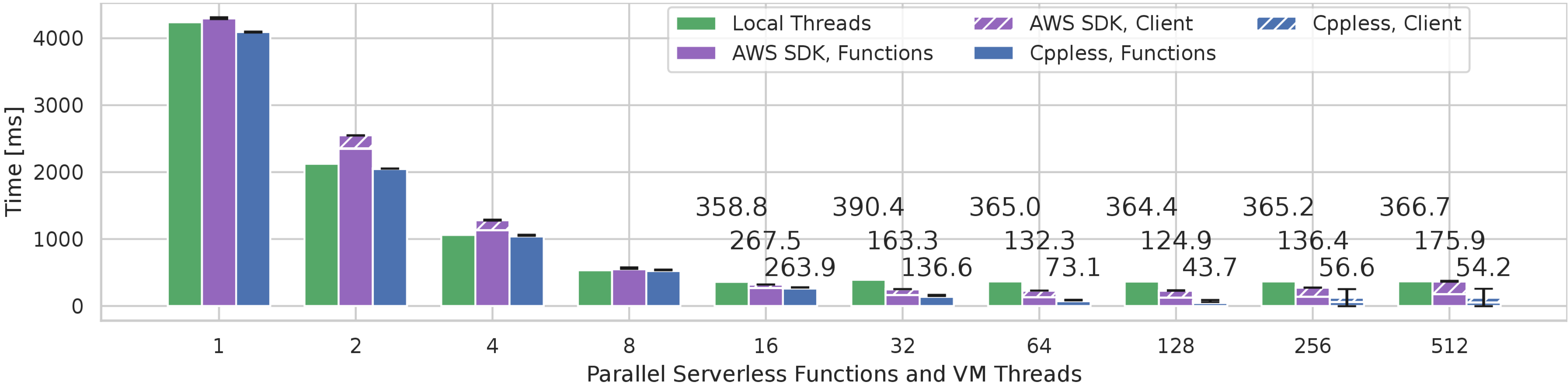
Median latency: 11-14 ms per invocation (< 64 workers)

Scalability limit: 230 ms at 2000 workers.

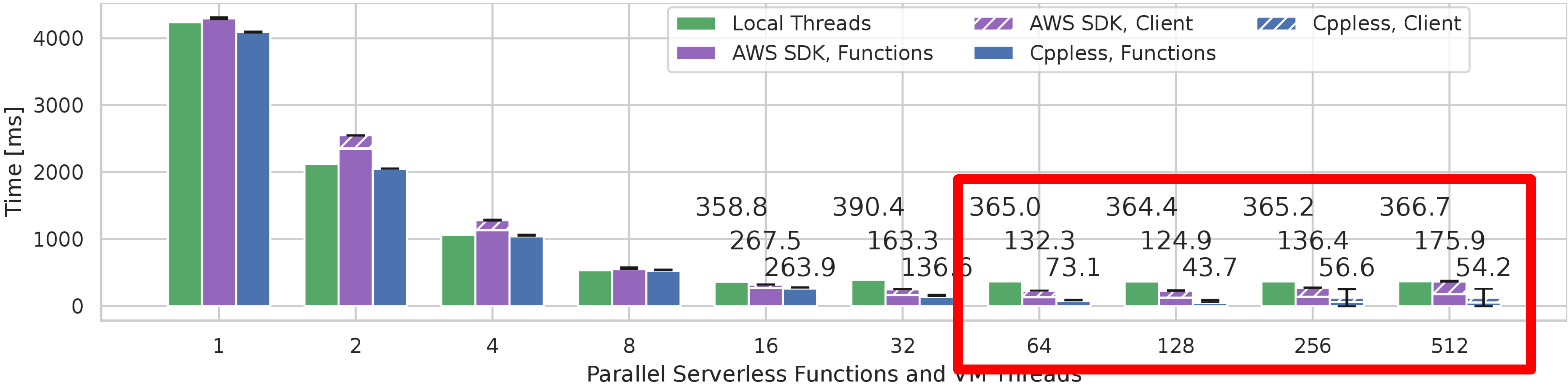
Evaluation: Monte Carlo PI 1



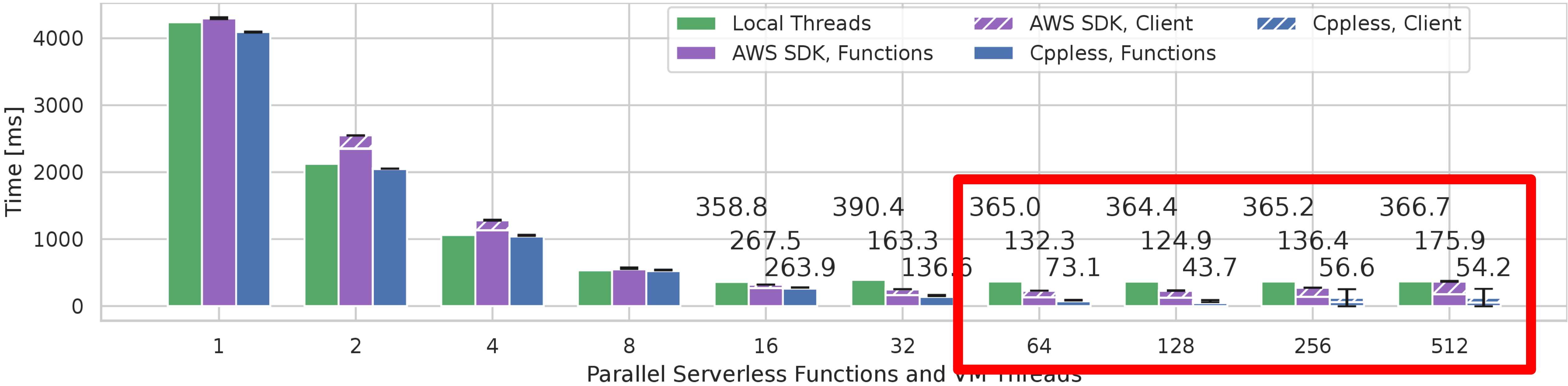
Evaluation: Monte Carlo PI 1



Evaluation: Monte Carlo PI 1



Evaluation: Monte Carlo PI 1



Dispatch performance: high-performance HTTP server is critical.

Evaluation: Cold Invocations 1

Measurement: cloud provider initialization time.

Evaluation: Cold Invocations 1

Measurement: cloud provider initialization time.

10 – 12 ms

Evaluation: Cold Invocations 1

Measurement: cloud provider initialization time.

10 – 12 ms

Python 3.8: **~120 ms**
Python 3.9+: **70-80 ms**

Evaluation: Cold Invocations 1

Measurement: cloud provider initialization time.

10 – 12 ms

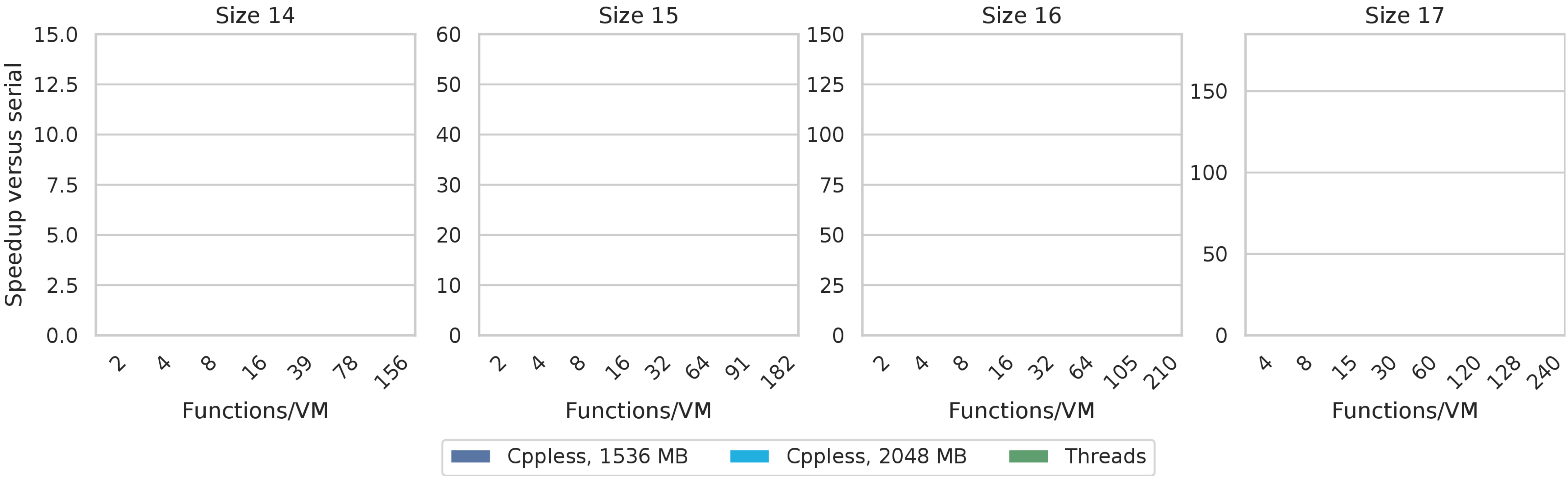
Python 3.8: **~120 ms**
Python 3.9+: **70-80 ms**

python™
+
NumPy

From **300 ms** to **620 ms**

Barcelona OpenMP Task Suite: NQueens 2

Local Threading: 35 lines of code
Cppless: 47 lines of code
Serialization: std::vector

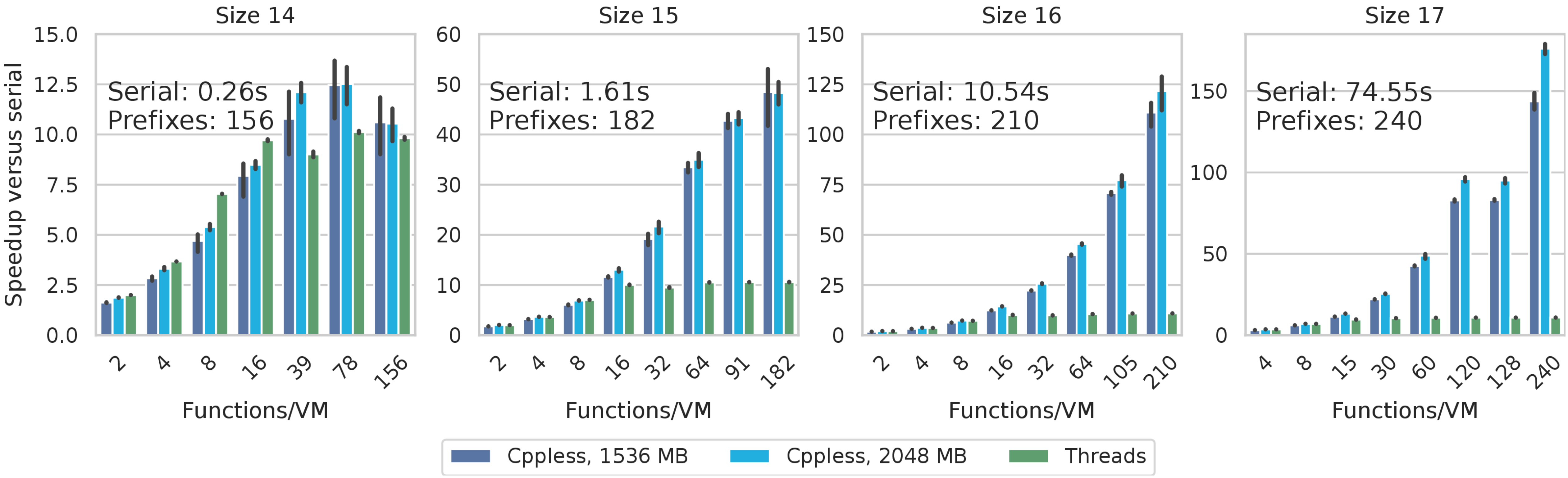


Barcelona OpenMP Task Suite: NQueens 2

Local Threading: 35 lines of code

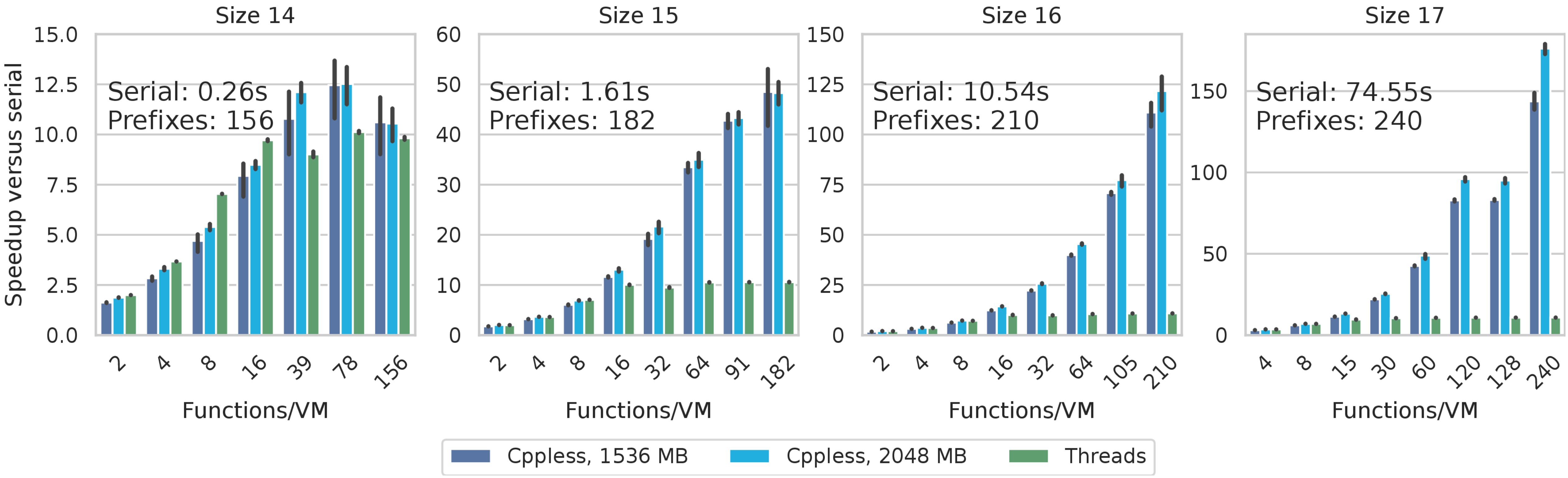
Cppless: 47 lines of code

Serialization: std::vector



Barcelona OpenMP Task Suite: NQueens 2

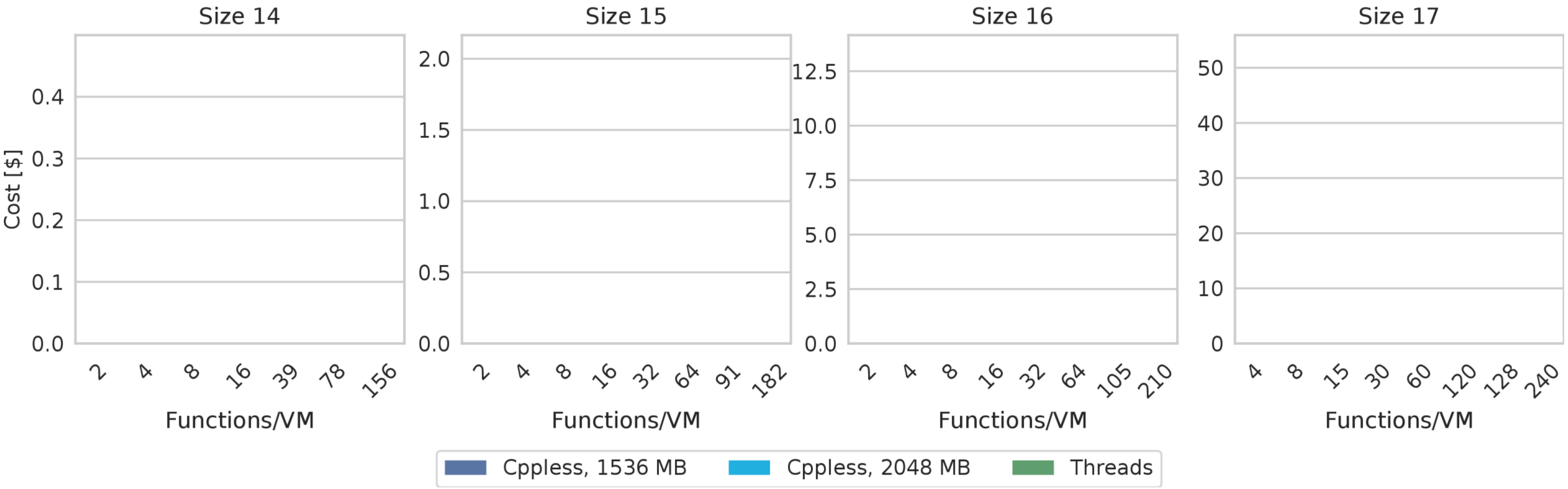
Local Threading: 35 lines of code
Cppless: 47 lines of code
Serialization: std::vector



Scalability limit: load imbalance between tasks.

Barcelona OpenMP Task Suite: NQueens 2

Local Threading: 35 lines of code
Cppless: 47 lines of code
Serialization: std::vector

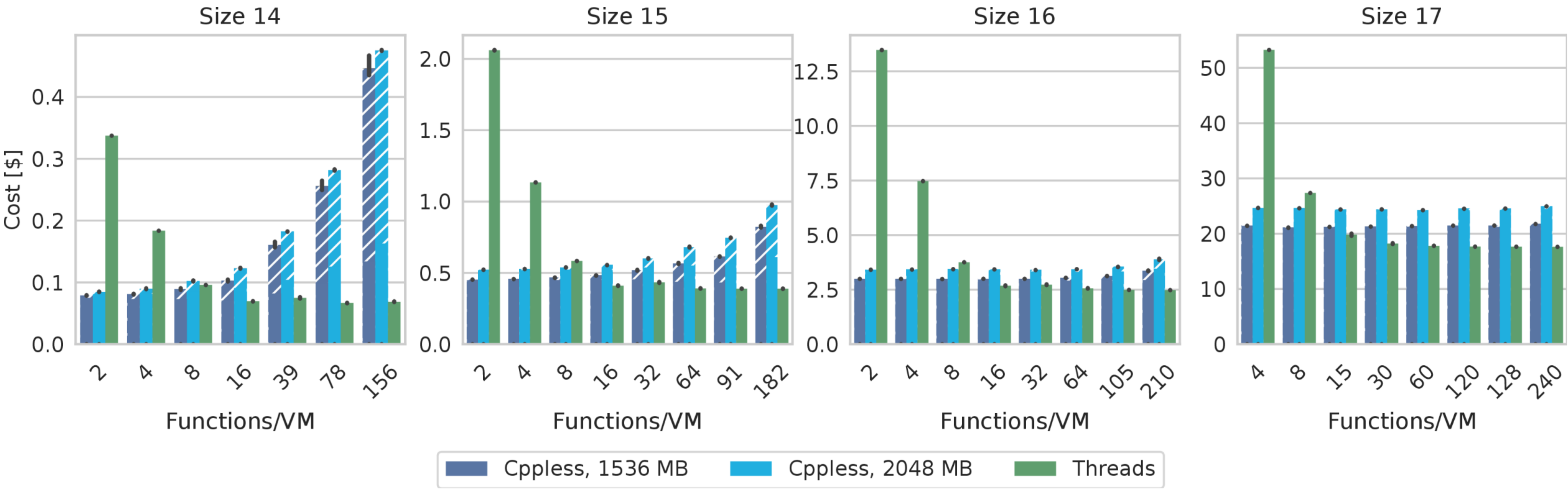


Barcelona OpenMP Task Suite: NQueens 2

Local Threading: 35 lines of code

Cppless: 47 lines of code

Serialization: std::vector

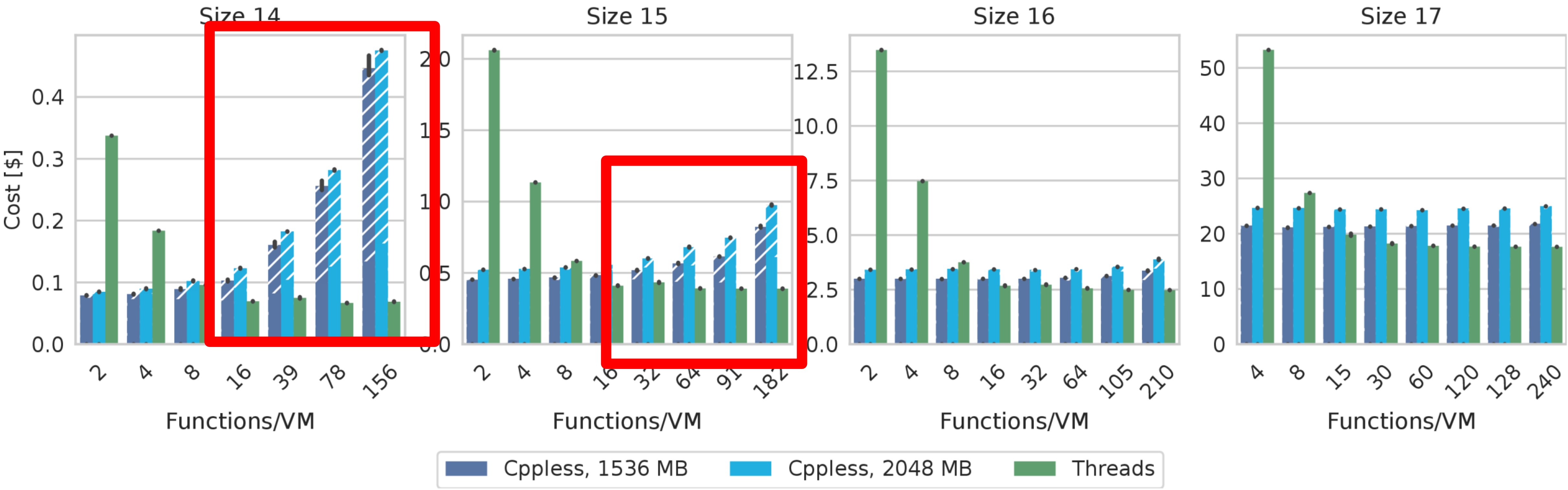


Barcelona OpenMP Task Suite: NQueens 2

Local Threading: 35 lines of code

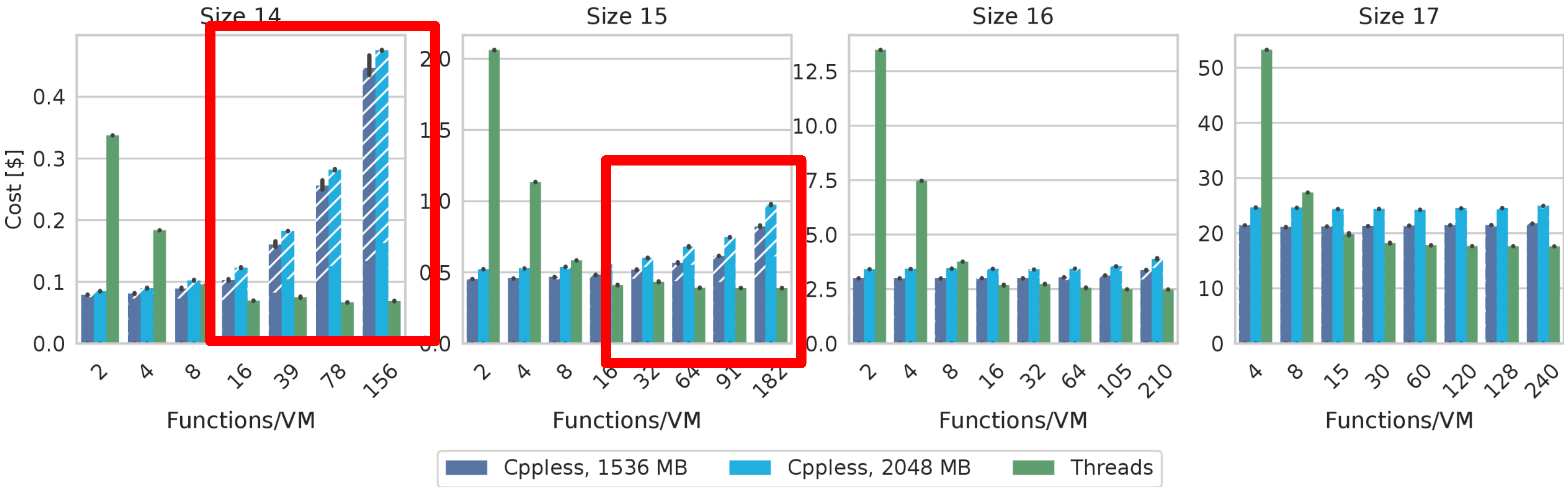
Cppless: 47 lines of code

Serialization: std::vector



Barcelona OpenMP Task Suite: NQueens 2

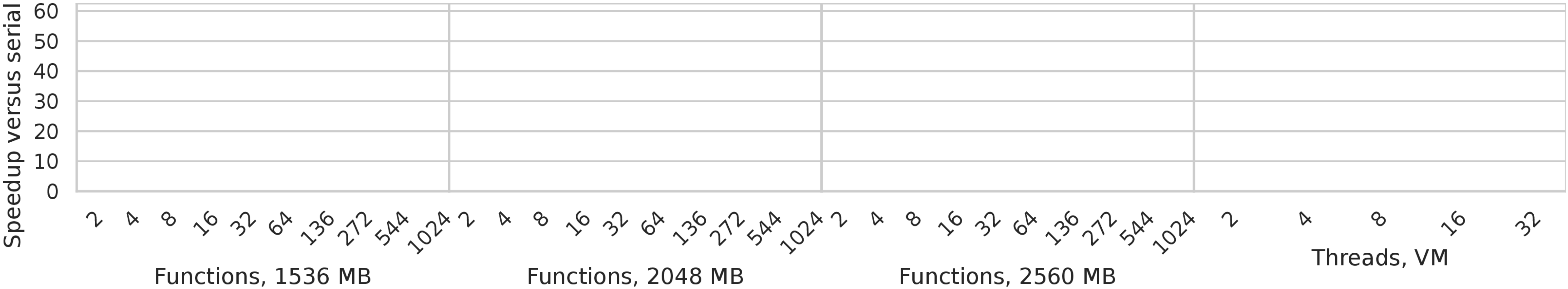
Local Threading: 35 lines of code
Cppless: 47 lines of code
Serialization: std::vector



Efficiency limit: impact of flat invocation fee on short-running tasks

Evaluation: Ray Tracing 3

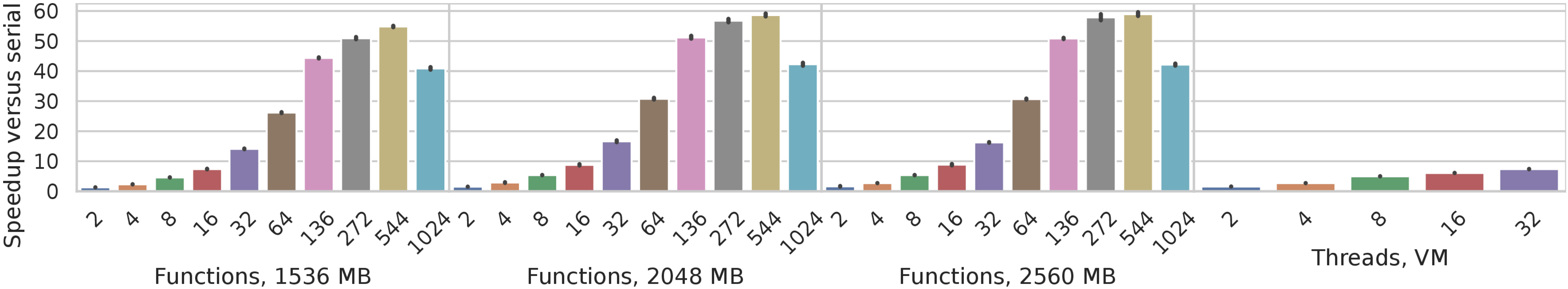
Size: 1,161 lines of code
Serialization: dynamic lists, shared pointers, polymorphic types.



Evaluation: Ray Tracing 3

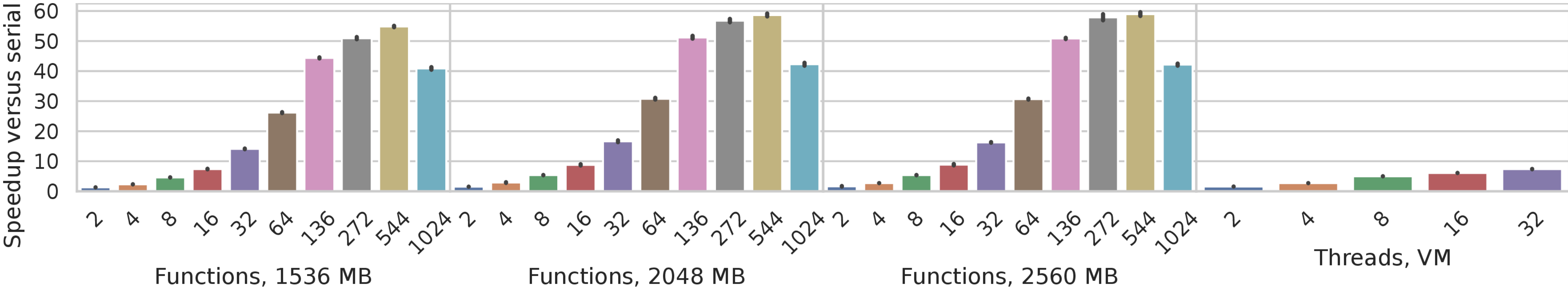
Size: 1,161 lines of code

Serialization: dynamic lists, shared pointers, polymorphic types.



Evaluation: Ray Tracing 3

Size: 1,161 lines of code
Serialization: dynamic lists, shared pointers, polymorphic types.



Scalability limit: load imbalance and serialization overhead at larger scale.

Still: 55x speedup at 6% increase in cost

High-Performance Solutions for Serverless

GitHub **spcl/cppless**

GitHub **spcl/cppless-artifact**

High-Performance Solutions for Serverless

GitHub spcl/cppless

GitHub spcl/cppless-artifact

spcl/serverless-benchmarks

spcl/rFaaS

spcl/PraaS

spcl/FaaSKeeper

spcl/XaaS

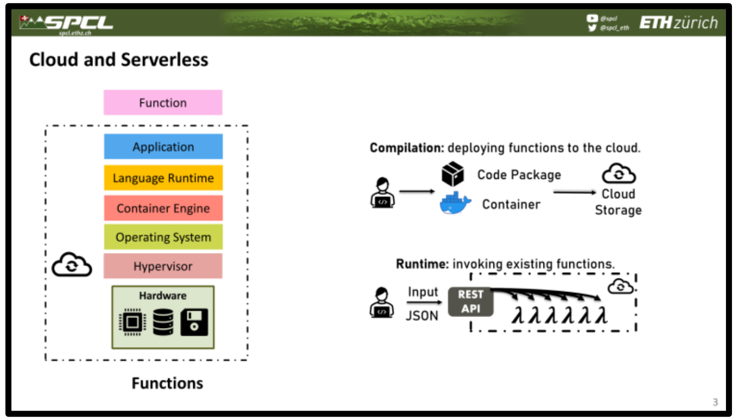
Conclusions

More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

Conclusions



Cloud and Serverless

Function

Application

Language Runtime

Container Engine

Operating System

Hypervisor

Hardware

Functions

Compilation: deploying functions to the cloud.

Code Package

Container

Cloud Storage

Runtime: invoking existing functions.

Input JSON

REST API

λλλλλλ

More of SPCL's research:

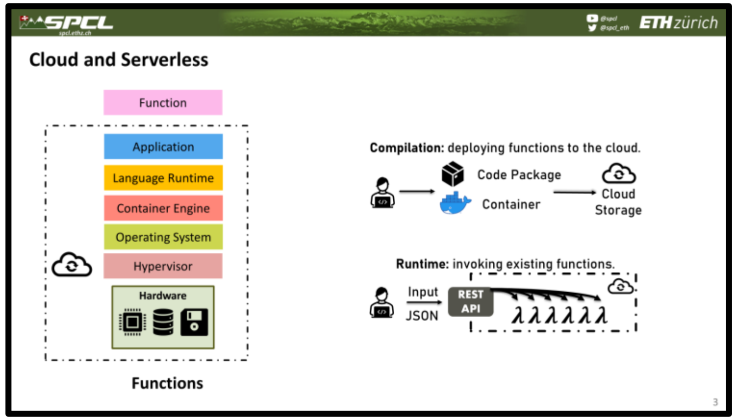
youtube.com/@spcl — 240+ Talks

twitter.com/spcl_eth — 1.7K+ Followers

github.com/spcl — 7.2K+ Stars

… or spcl.ethz.ch

Conclusions



Cloud and Serverless

Function

Application

Language Runtime

Container Engine

Operating System

Hypervisor

Hardware

Functions

Compilation: deploying functions to the cloud.

Code Package

Container

Cloud Storage

Runtime: invoking existing functions.

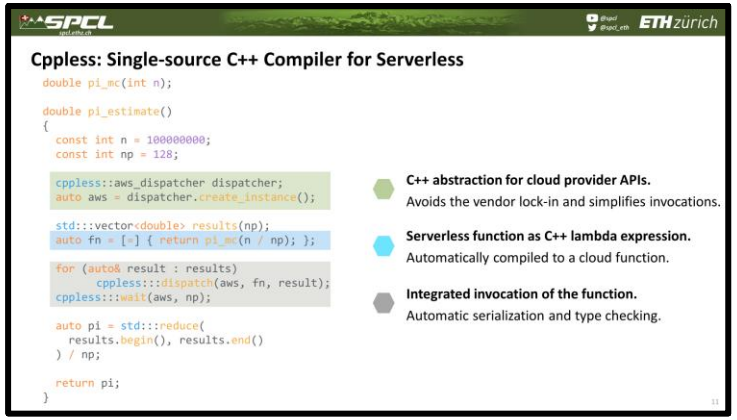
Input

JSON

REST API

λλλλλλ

3



Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std:::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless:::dispatch(aws, fn, result);
  cppless:::wait(aws, np);

  auto pi = std:::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.
Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.
Automatically compiled to a cloud function.

Integrated invocation of the function.
Automatic serialization and type checking.

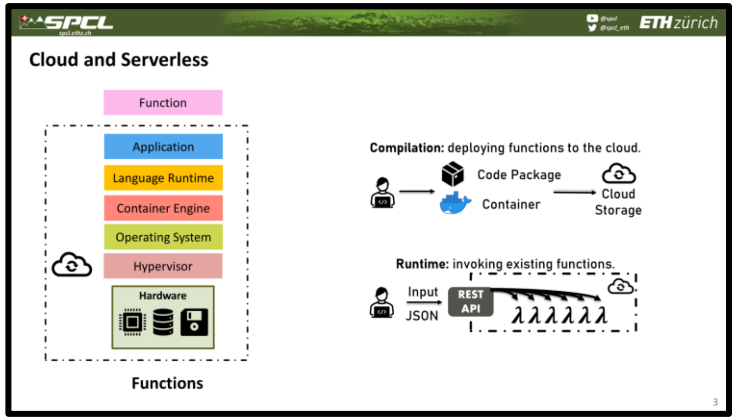
11

More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

Conclusions



Cloud and Serverless

Function

Application

Language Runtime

Container Engine

Operating System

Hypervisor

Hardware

Functions

Compilation: deploying functions to the cloud.

Code Package

Container

Cloud Storage

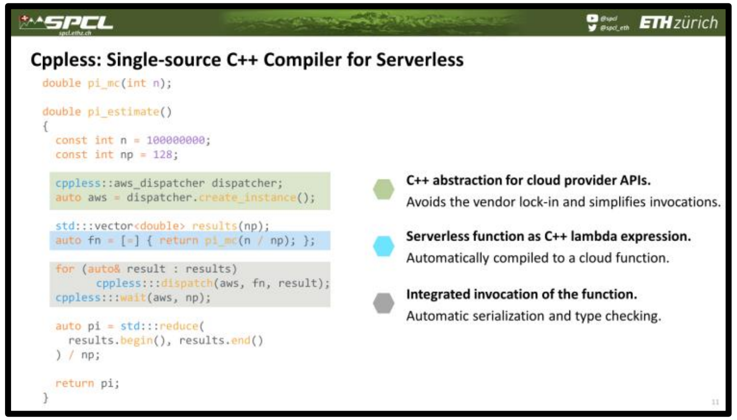
Runtime: invoking existing functions.

Input

JSON

REST API

λλλλλλ



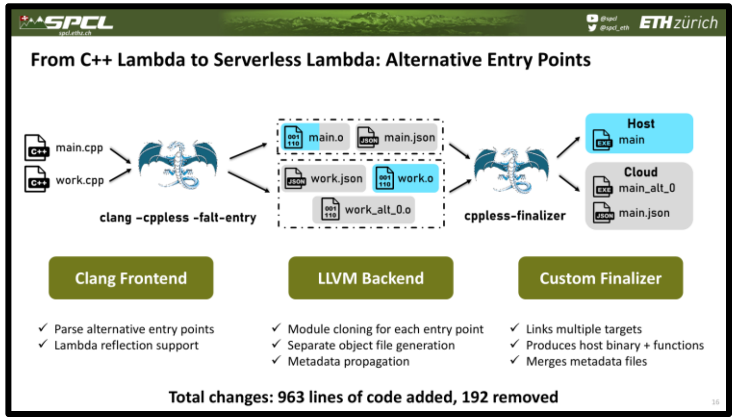
Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std:::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless:::dispatch(aws, fn, result);
  cppless:::wait(aws, np);

  auto pi = std:::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.
Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.
Automatically compiled to a cloud function.

Integrated invocation of the function.
Automatic serialization and type checking.



From C++ Lambda to Serverless Lambda: Alternative Entry Points

Clang Frontend

- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend

- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer

- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

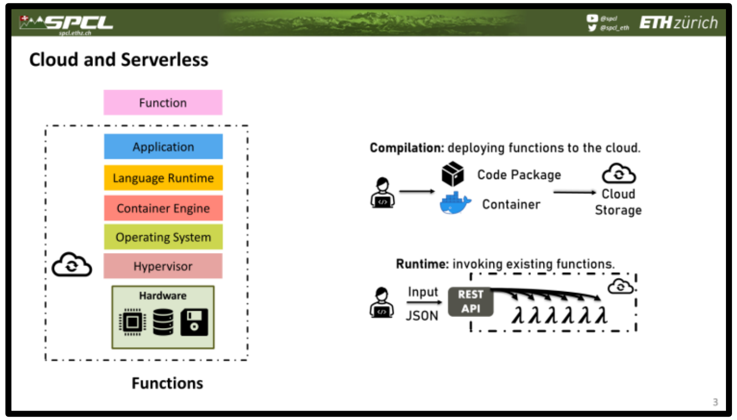
Total changes: 963 lines of code added, 192 removed

More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

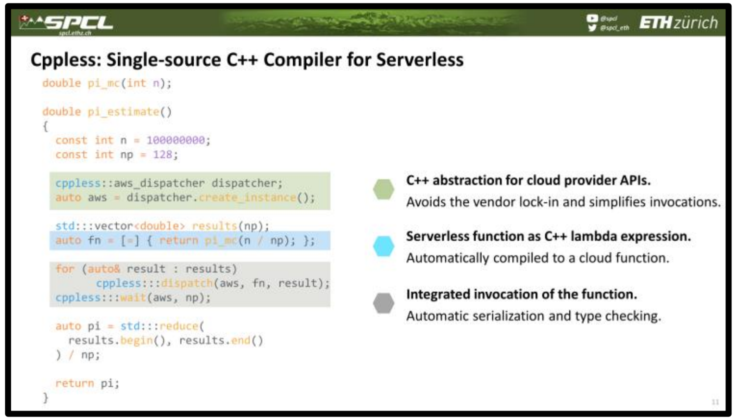
Conclusions



Cloud and Serverless

Compilation: deploying functions to the cloud.

Runtime: invoking existing functions.



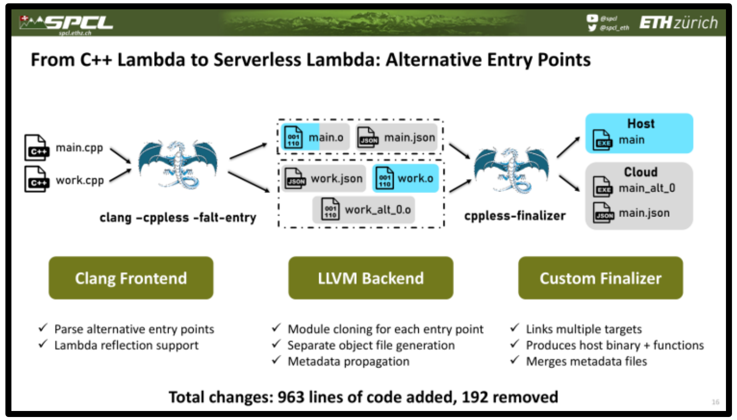
Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std:::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless:::dispatch(aws, fn, result);
  cppless:::wait(aws, np);

  auto pi = std:::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.
Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.
Automatically compiled to a cloud function.

Integrated invocation of the function.
Automatic serialization and type checking.



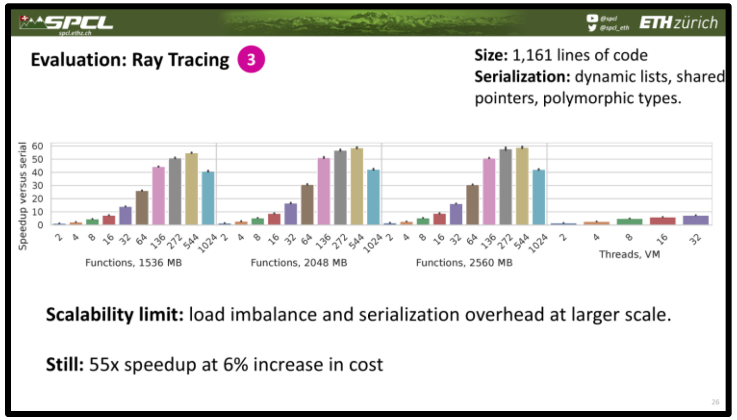
From C++ Lambda to Serverless Lambda: Alternative Entry Points

Clang Frontend
- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend
- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer
- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

Total changes: 963 lines of code added, 192 removed



Evaluation: Ray Tracing 3

Size: 1,161 lines of code
Serialization: dynamic lists, shared pointers, polymorphic types.

Scalability limit: load imbalance and serialization overhead at larger scale.

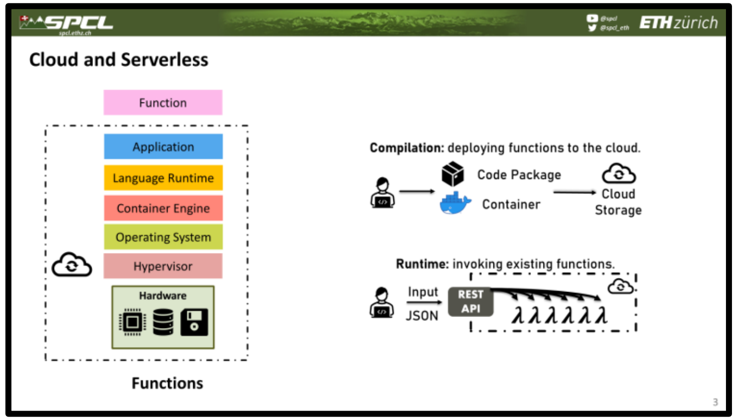
Still: 55x speedup at 6% increase in cost

More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

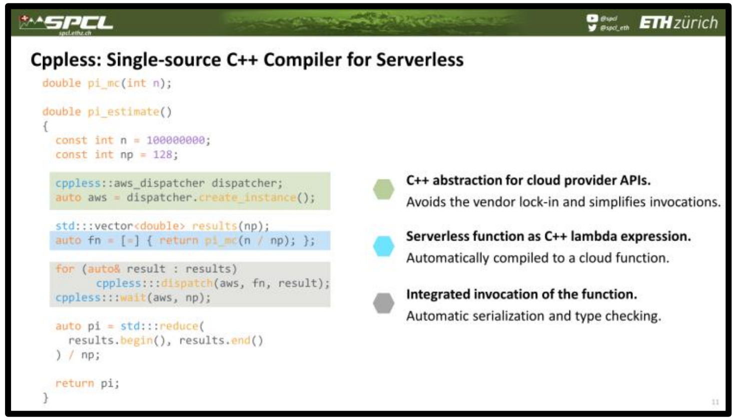
Conclusions



Cloud and Serverless

Compilation: deploying functions to the cloud.

Runtime: invoking existing functions.



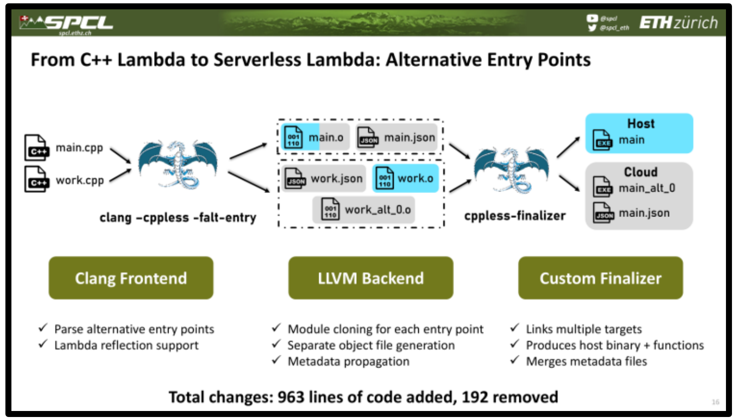
Cppless: Single-source C++ Compiler for Serverless

```
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std:::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless:::dispatch(aws, fn, result);
  cppless:::wait(aws, np);

  auto pi = std:::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.
Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.
Automatically compiled to a cloud function.

Integrated invocation of the function.
Automatic serialization and type checking.



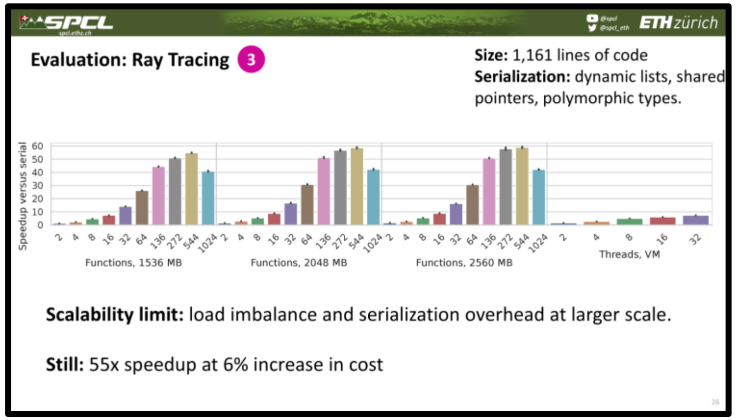
From C++ Lambda to Serverless Lambda: Alternative Entry Points

Clang Frontend
- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend
- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer
- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

Total changes: 963 lines of code added, 192 removed



Evaluation: Ray Tracing 3

Size: 1,161 lines of code
Serialization: dynamic lists, shared pointers, polymorphic types.

Scalability limit: load imbalance and serialization overhead at larger scale.

Still: 55x speedup at 6% increase in cost

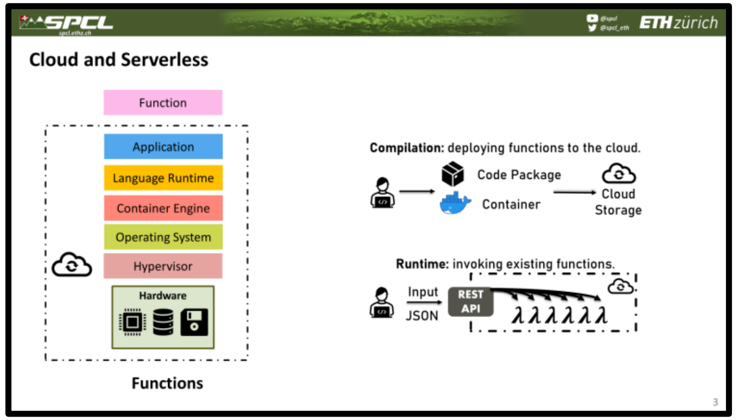
More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

Cppless

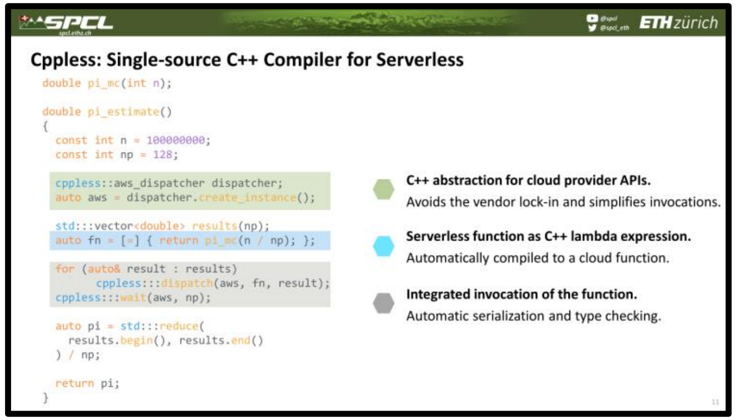
Conclusions



Cloud and Serverless

Compilation: deploying functions to the cloud.

Runtime: invoking existing functions.



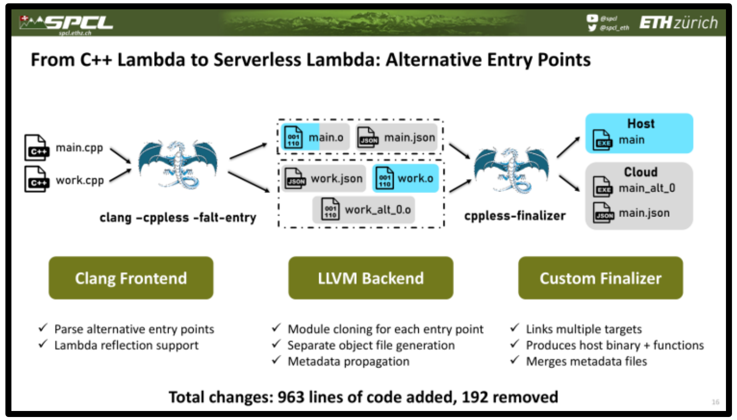
Cppless: Single-source C++ Compiler for Serverless

```cpp
double pi_mc(int n);

double pi_estimate()
{
  const int n = 100000000;
  const int np = 128;

  cppless::aws_dispatcher dispatcher;
  auto aws = dispatcher.create_instance();

  std:::vector<double> results(np);
  auto fn = [=] { return pi_mc(n / np); };

  for (auto& result : results)
        cppless:::dispatch(aws, fn, result);
  cppless:::wait(aws, np);

  auto pi = std:::reduce(
    results.begin(), results.end()
  ) / np;

  return pi;
}
```

C++ abstraction for cloud provider APIs.
Avoids the vendor lock-in and simplifies invocations.

Serverless function as C++ lambda expression.
Automatically compiled to a cloud function.

Integrated invocation of the function.
Automatic serialization and type checking.



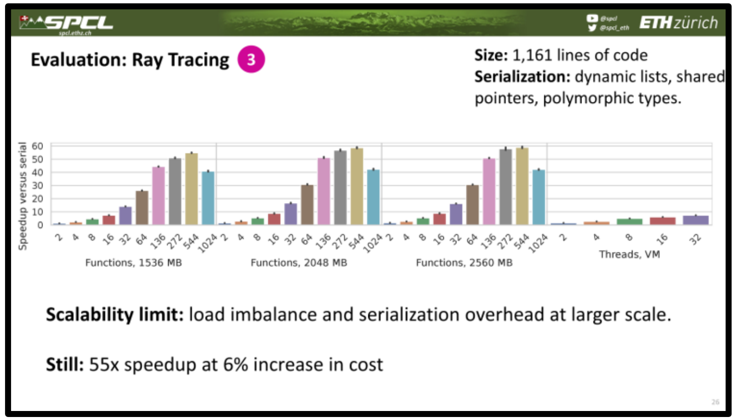
From C++ Lambda to Serverless Lambda: Alternative Entry Points

Clang Frontend
- ✓ Parse alternative entry points
- ✓ Lambda reflection support

LLVM Backend
- ✓ Module cloning for each entry point
- ✓ Separate object file generation
- ✓ Metadata propagation

Custom Finalizer
- ✓ Links multiple targets
- ✓ Produces host binary + functions
- ✓ Merges metadata files

Total changes: 963 lines of code added, 192 removed



Evaluation: Ray Tracing 3

Size: 1,161 lines of code
Serialization: dynamic lists, shared pointers, polymorphic types.

Scalability limit: load imbalance and serialization overhead at larger scale.

Still: 55x speedup at 6% increase in cost

More of SPCL's research:

youtube.com/@spcl	240+ Talks
twitter.com/spcl_eth	1.7K+ Followers
github.com/spcl	7.2K+ Stars

… or spcl.ethz.ch

Cppless

Serverless Projects

Evaluation: Binary Size