C++ now

2025

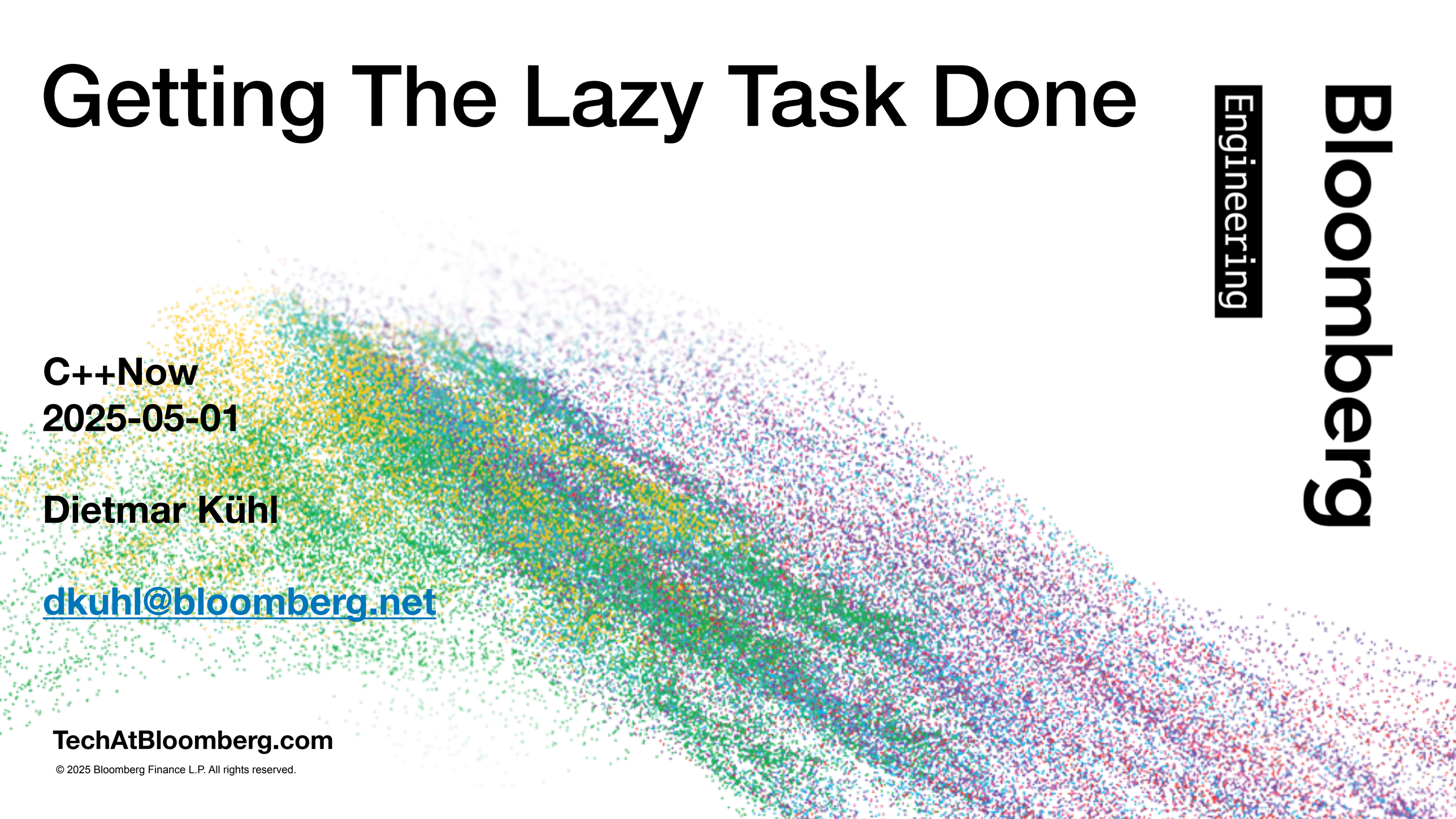
Getting the Lazy Task Done

Dietmar Kühl

Getting The Lazy Task Done

Bloomberg

Engineering



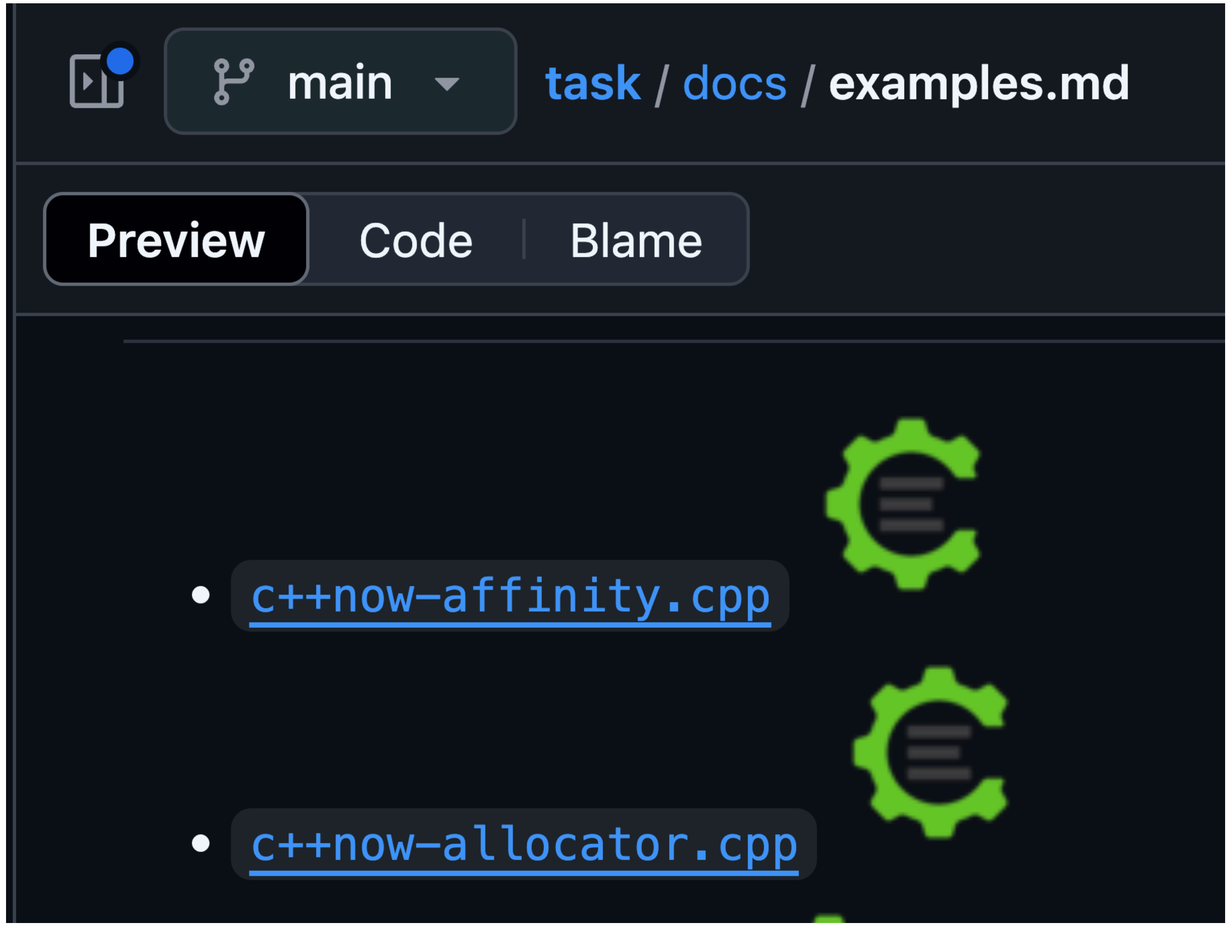
C++Now
2025-05-01

Dietmar Kühl

dkuhl@bloomberg.net

TechAtBloomberg.com

Live Code Examples



Objective

- std::execution is bound to become part of C++26
- users most likely want to use coroutines to create senders
- create a coroutine task type for the standard C++ library

Bloomberg

Engineering

Motivation

- Composing work from algorithms is possible but unfamiliar
  - Writing a function with some co_* keywords is more familiar
  - Writing a coroutine is often a lot easier
- Coroutines provide a type-erased interface

Sender/Receiver in a Nutshell

- sender: represent work
  S::comp_signs<set_value_t(v…), set_error(e), set_stopped>
  s.connect(receiver)

- receiver: consume results from a sender
  r.set_value(v…), r.set_error(e), r.set_stopped()
  r.get_env()

- operation state = connect(sender, receiver)
  o.start()

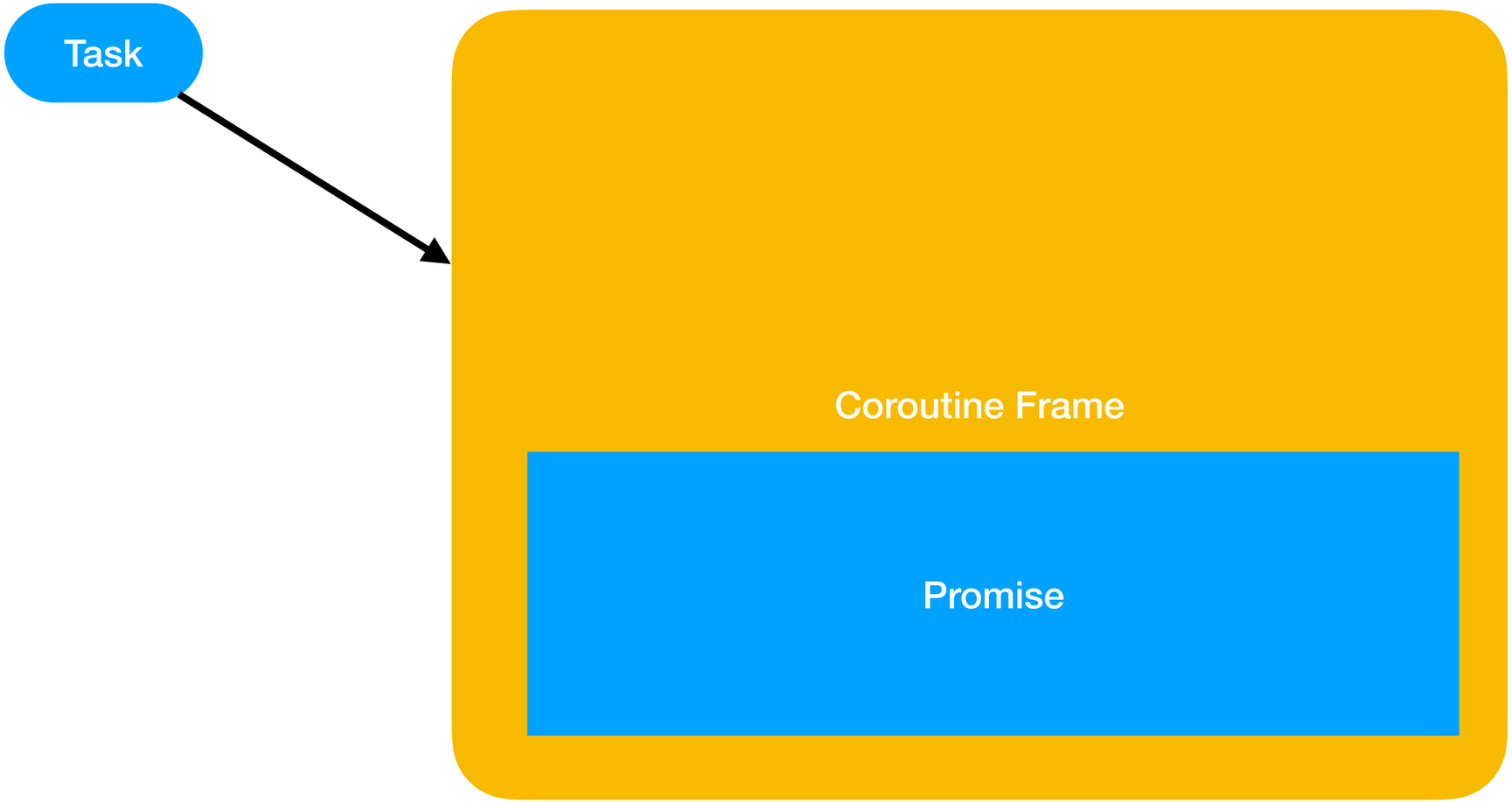
Basics

```
#include <execution>

std::execution::task<> basic() {
    co_await std::suspend_never{};
}



int main() { std::execution::sync_wait(basic()); }
```

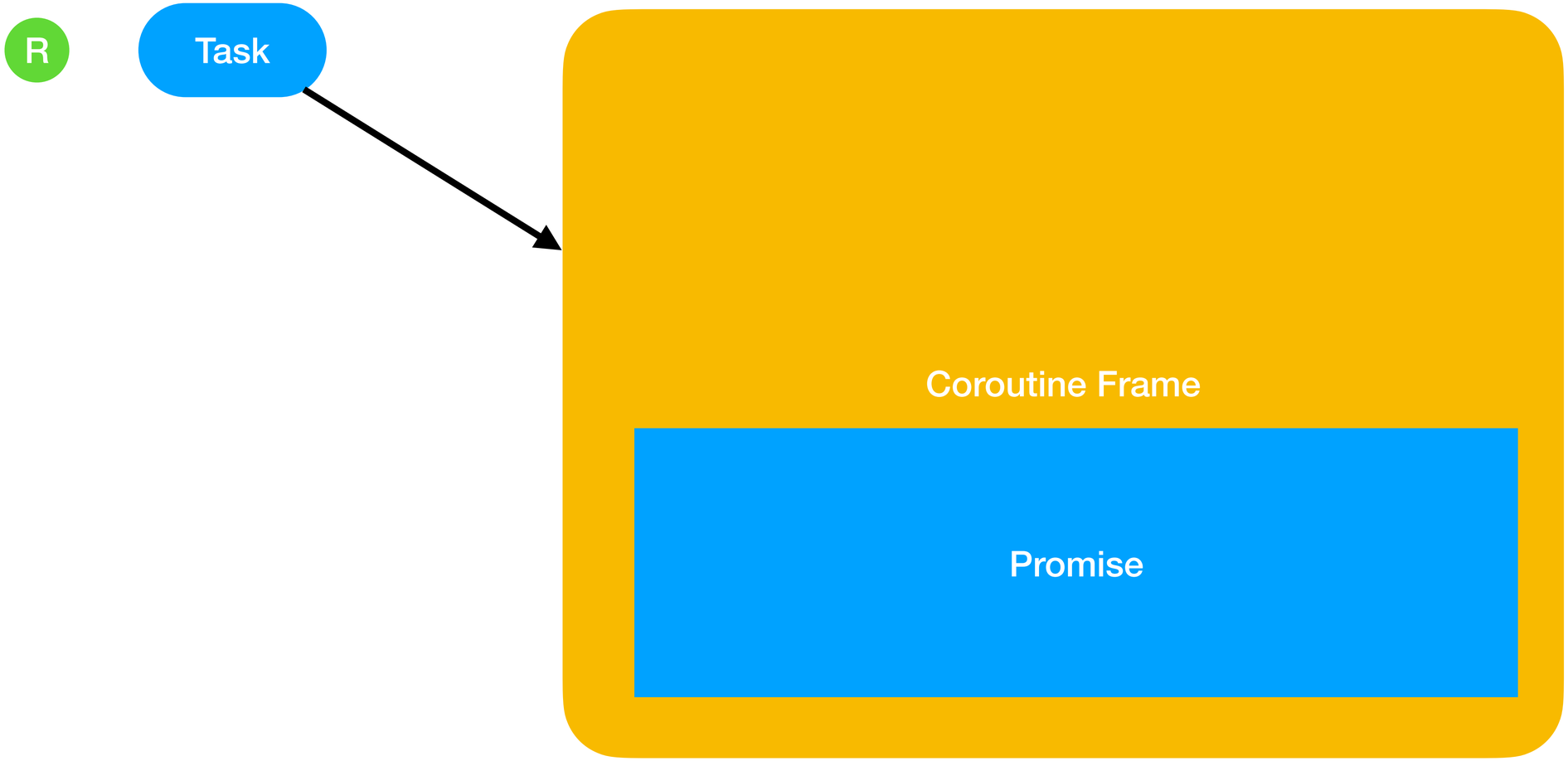
Basics

```
#include <execution>

std::execution::task<> basic() {
    co_await std::execution::just();
}



int main() { std::execution::sync_wait(basic()); }
```

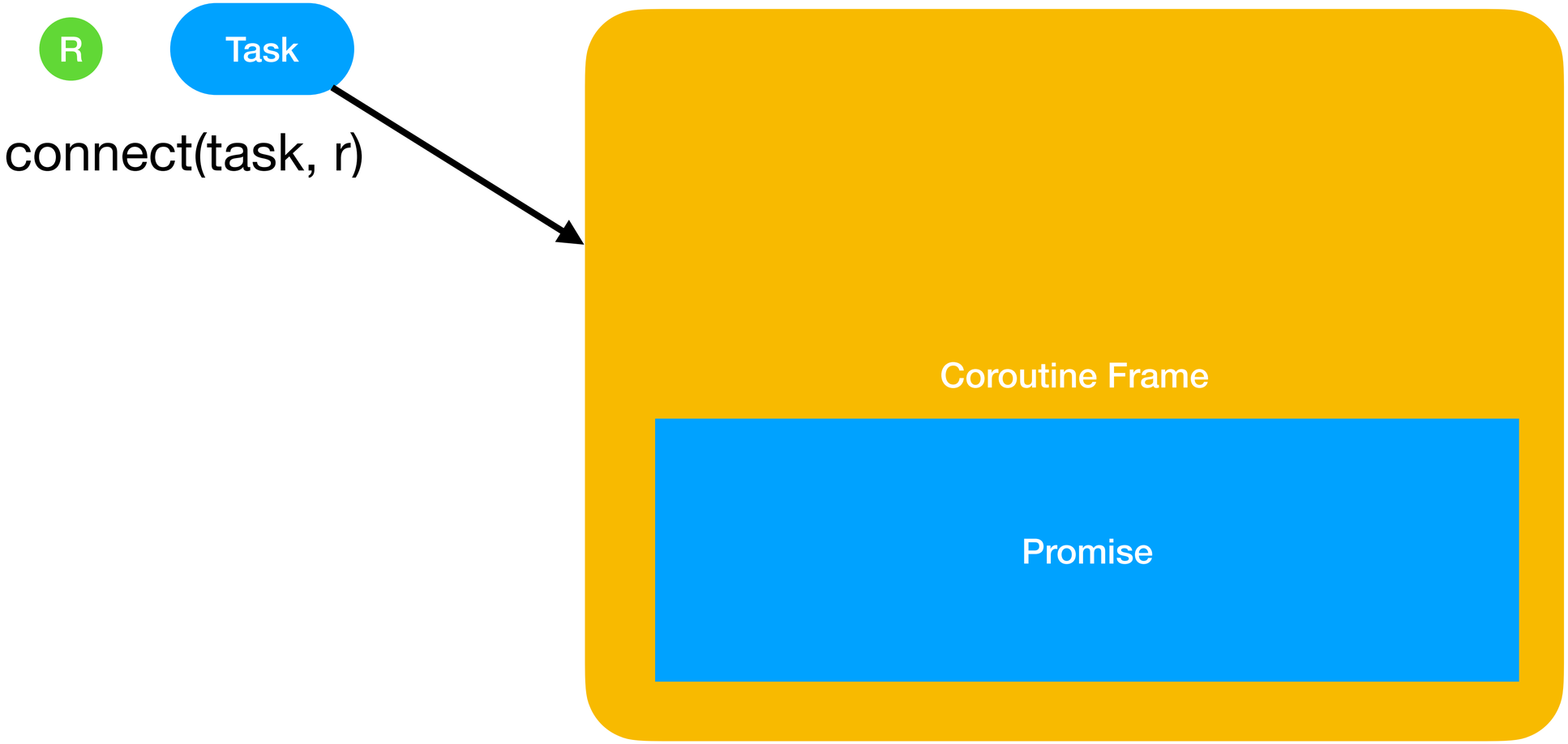

Task is a Sender



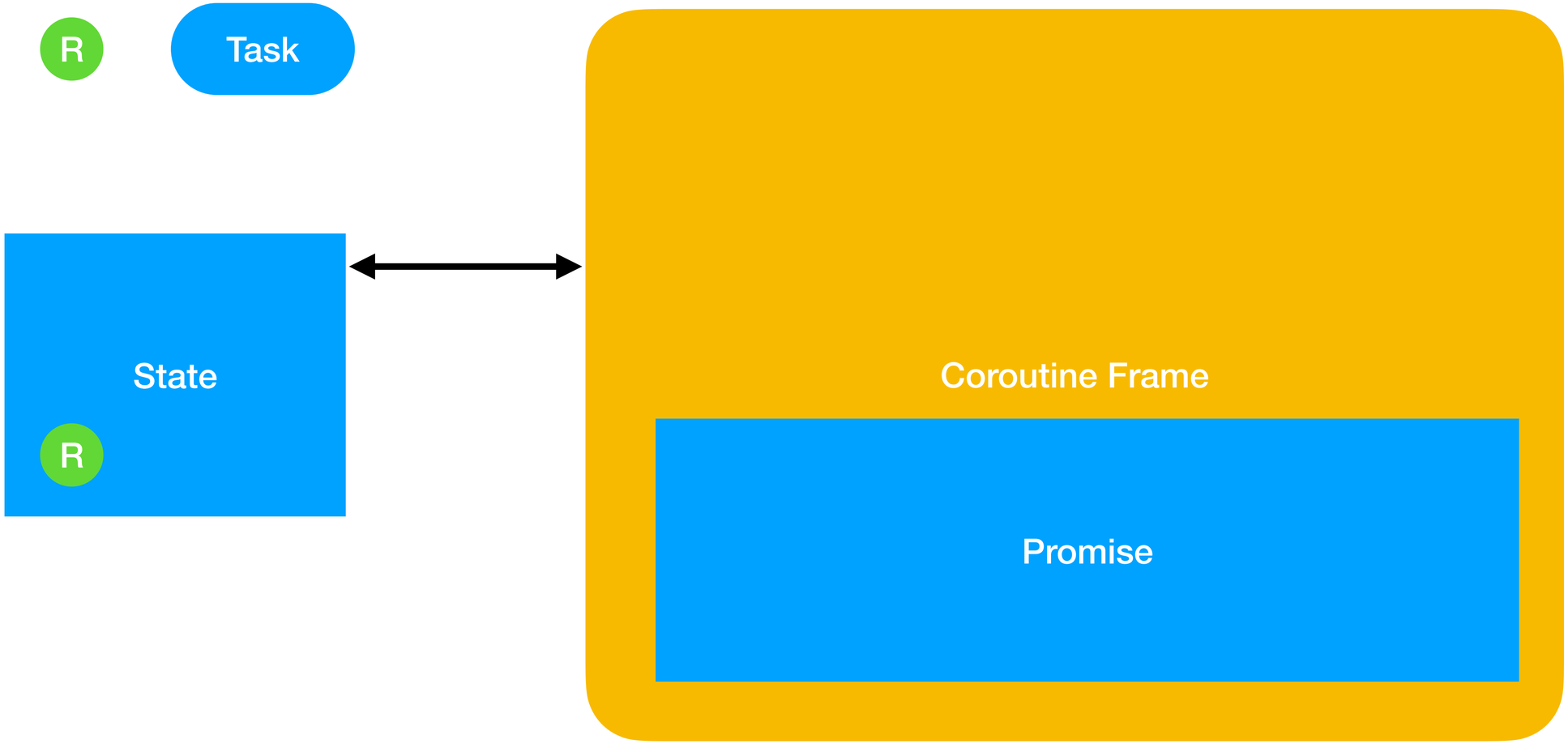
Task is a Sender



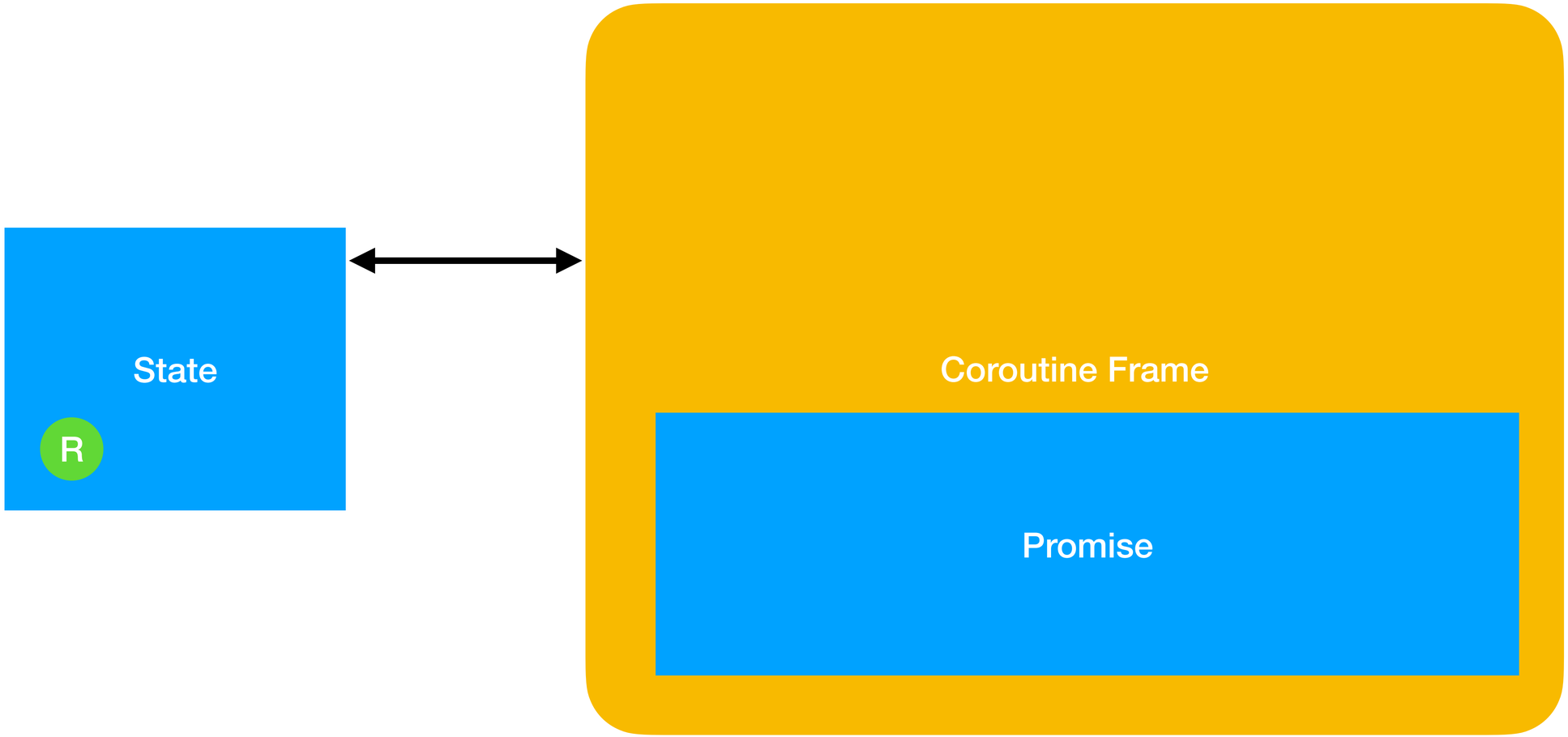
Task is a Sender



Task is a Sender



Task is a Sender



Result Types

```
#include <execution>

std::execution::task<> result_types() {
    /* void */        co_await std::execution::just();
    int              n = co_await std::execution::just(17);
    std::tuple  t = co_await std::execution::just(17, true);
}

int main() { std::execution::sync_wait(result_types()); }
```


Bloomberg

Engineering

Errors

```
#include <execution>

std::execution::task<> error_result() {
    try { co_await std::execution::just_error(17); }
    catch (int n) { std::print("Error: {}\n", n); }
}


int main() { std::execution::sync_wait(error_result()); }
```

Errors Without Exception

```
#include <execution>

std::execution::task<> error_result() {
    auto e = co_await as_expected(std::execution::just(17)); }
    auto u = co_await as_expected(std::execution::just_error(17)); }
}

int main() { std::execution::sync_wait(error_result()); }
```

not in the C++ standard library

Cancellation

```
#include <execution>

std::execution::task<> cancel() {
    co_await std::execution::just_stopped(); // never resumes
}



int main() { std::execution::sync_wait(cancel()); }
```


as_awaitable(sender, promise)

- The results of awaiting expressions in a task come from as_awaitable()
- std::execution::as_awaitable(sender, promise) came from P2300
- Should there be additional/different transformations?
  - Problem: everybody wants their specific transformations
  - Conclusion: the transformations should be explicit

Task’s Return Type

```
#include <execution>

std::execution::task<> result() {
    co_return;
}

int main() {
    std::execution::sync_wait(result());
}
```

Task’s Return Type

```
#include <execution>

std::execution::task<void> result() {
    co_return;
}

int main() {
    std::execution::sync_wait(result());
}
```

Task's Return Type

```
#include <execution>

std::execution::task<int> result() {
    co_return 17;
}

int main() {
    auto[n] = *std::execution::sync_wait(result());
}
```

Task’s Completion Signatures

task<void>

- std::execution::set_value_t()
- std::execution::set_error_t(std::exception_ptr)
- std::execution::set_stopped_t()

Task’s Completion Signatures

task<T>

- std::execution::set_value_t(T)
- std::execution::set_error_t(std::exception_ptr)
- std::execution::set_stopped_t()

Task's Completion Signatures

```
struct context {
    using error_types = comp_sigs<set_error_t(E), set_error_t(F)>;
};
task<T, context>
```

- std::execution::set_value_t(T)
- std::execution::set_error_t(E), std::execution::set_error_t(F)
- std::execution::set_stopped_t()

Task's Completion Signatures

```
struct context {
    using error_types = comp_sigs<set_error_t(E), set_error_t(F)>;
};
task<T, context>
```

No set_error_t(exception_ptr) but an exception ⇒ terminate().

- It can't be statically determined if an exception may be thrown.
- context::error_types can be empty ⇒ no error completions.

Completing With An Error

```
struct ctxt { using error_types = comp_sigs<set_error_t(int)>; };

task<int, ctxt> call(int v) {
    if (v == 1) co_yield with_error(-1);
    co_return 2 * v;
}
int main(int ac, char*) {
    auto[n] = *sync_wait(
        call(ac) | upon_error([](int e){ print("error({})\n", e); return -1; }));
}
```

Why Use co_yield?

- co_return with_error(e) can’t be used for coroutines returning void:
  - promise can’t have both return_value(T) and return_void().
- co_await would work but is used to await outstanding work.
- co_yield conceptually yields a result.
- Other than that, only exceptions would work.
  - The whole point is to allow avoiding exceptions.

Allocator Support

- Memory for the coroutine may need to be allocated
- Let C fun(A&&…) be the signature used to create a coroutine
- There are two helpful interfaces which are used when defined:
  - C::promise_type::operator new(size_t, A&&…)
  - C::promise_type::promise_type(A&&…)

Allocator Support Interface

1. context::allocator_type can be used to define an allocator type
   - By default the type is std::allocator<std::byte>
2. If there is an std::allocator_arg_t parameter
   - use the next parameter to initialise an allocator_type object
3. Use the allocator_type object to manage memory

std::destroying_delete_t

- operator delete(T* ptr, std::destroying_delete_t):
  - *ptr needs to be destroyed and memory needs to be released
- The allocator can be retrieved from *ptr before destroying *ptr
- Sadly, doesn’t work for coroutine promise_types!
  - The allocated object has promise_type just as a member

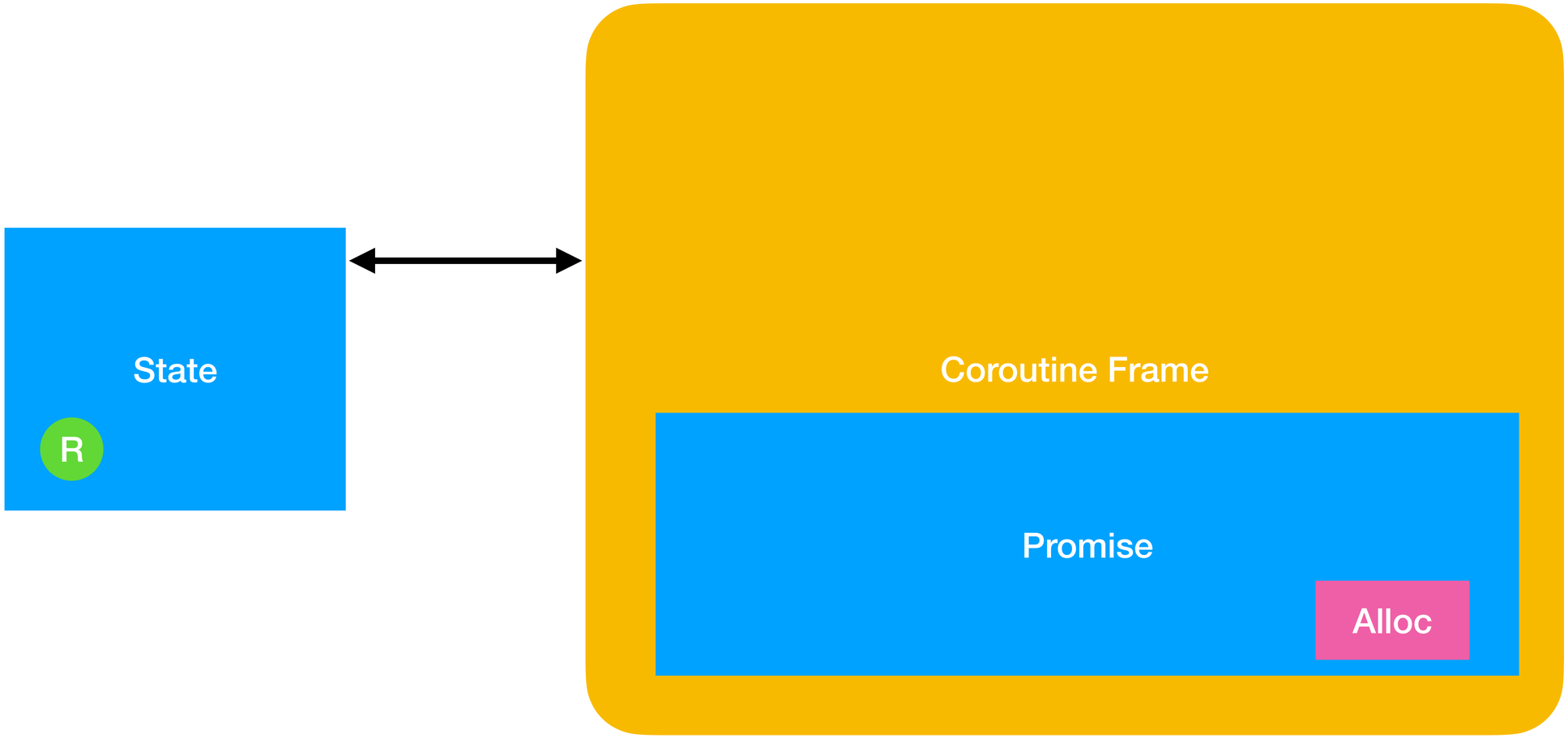
Allocator Example

```
struct with_alloc {
    using allocator_type = pmr::polymorphic_allocator<std::byte>;
};
task<int, with_alloc> coro(int value, auto&&…) {
    co_await just(value);
}
int main() {
    sync_wait(coro(0));
    sync_wait(coro(1, allocator_arg, pmr::new_delete_resource()));
}
```

Allocator From Environment

```
struct with_alloc {
    using allocator_type = pmr::polymorphic_allocator<std::byte>;
};
task<int, with_alloc> coro(int value, auto&&…) {
    auto a = co_await execution::read_env(execution::get_allocator);
}
int main() {
    sync_wait(coro(0));
    sync_wait(coro(1, allocator_arg, pmr::new_delete_resource()));
}
```

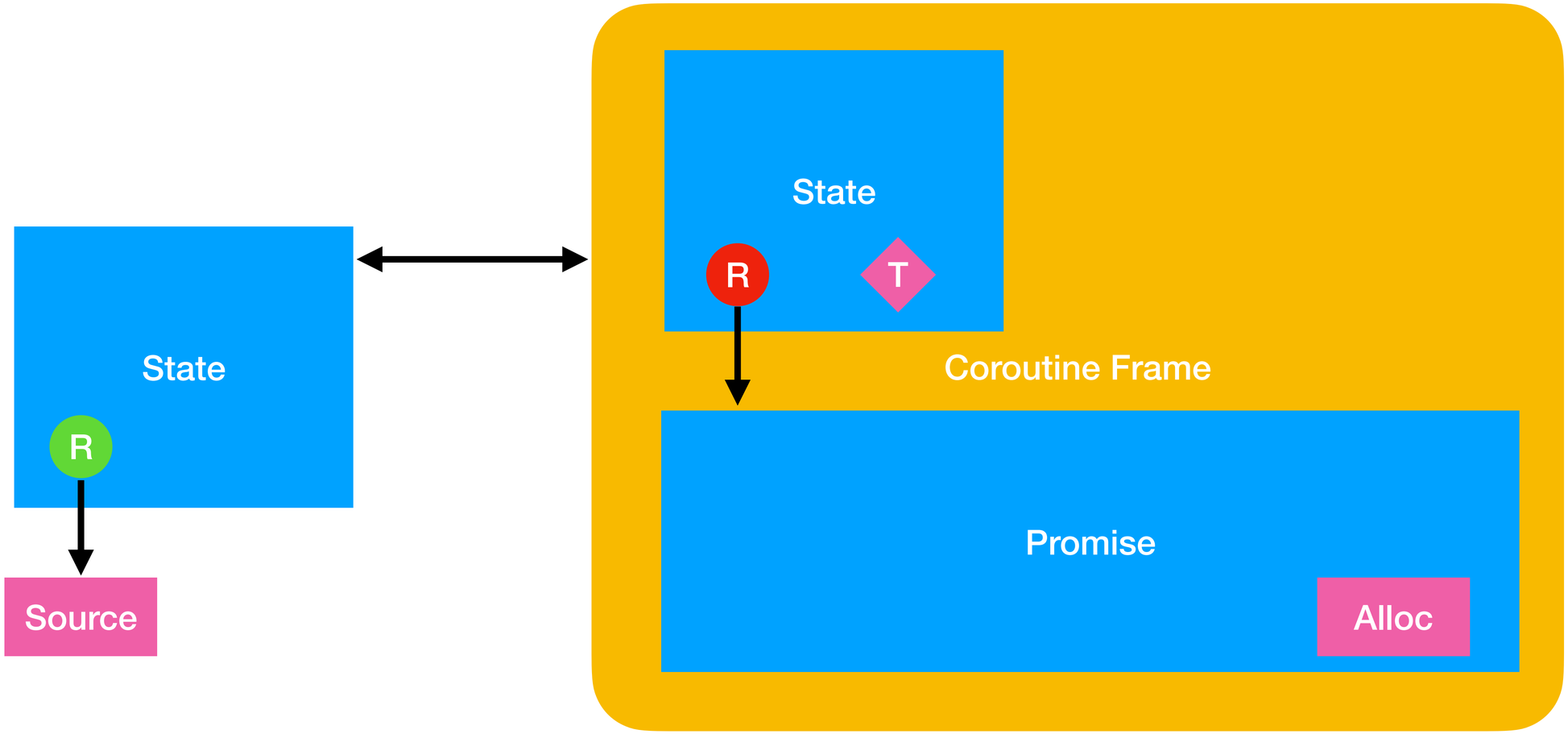
Task Holds an Allocator



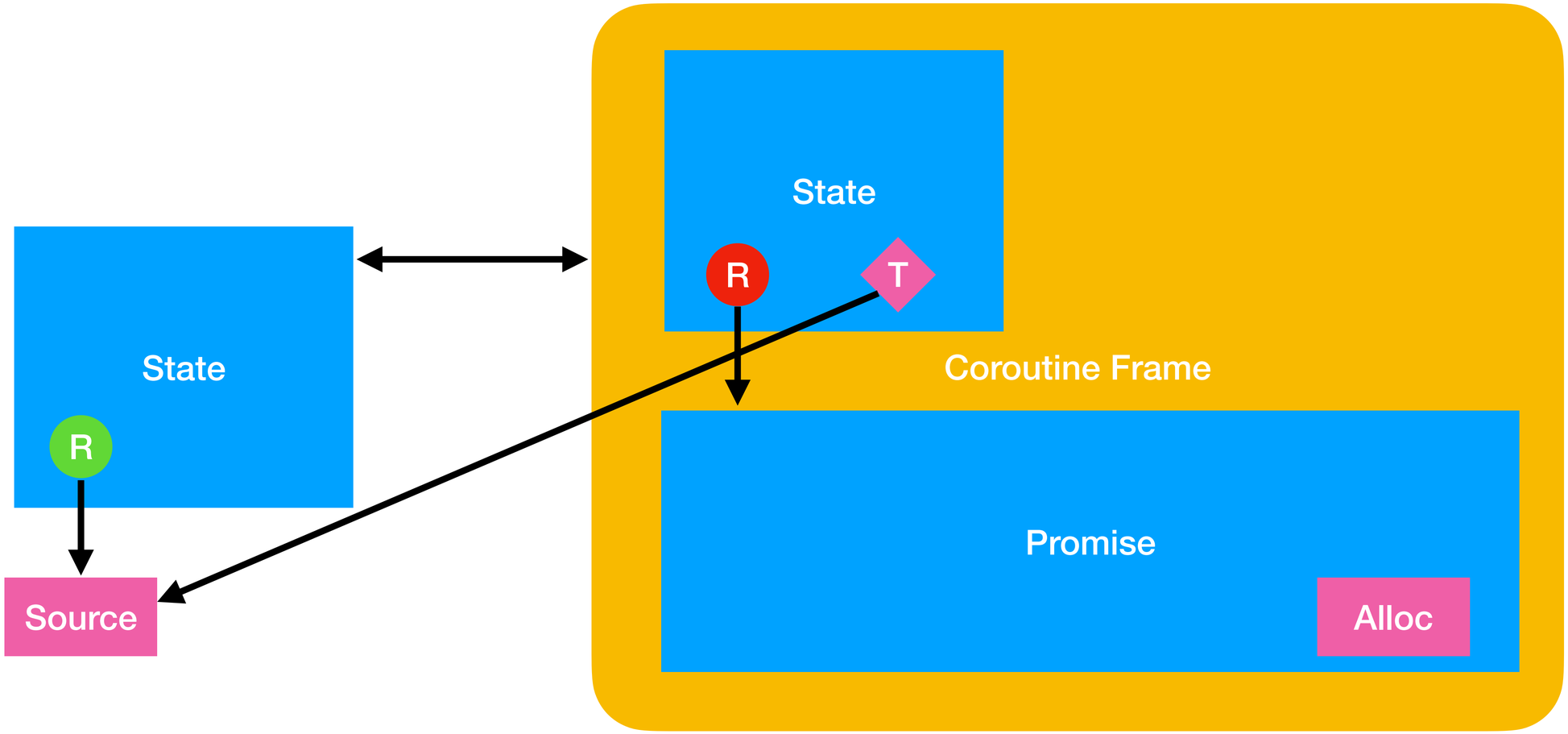
Stop Token From Environment

```
task<> stopping() {
    auto token = co_await read_env(get_stop_token);
    std::size_t count{};
    while (not token.stop_requested()) {
        ++count;
    }
}
```

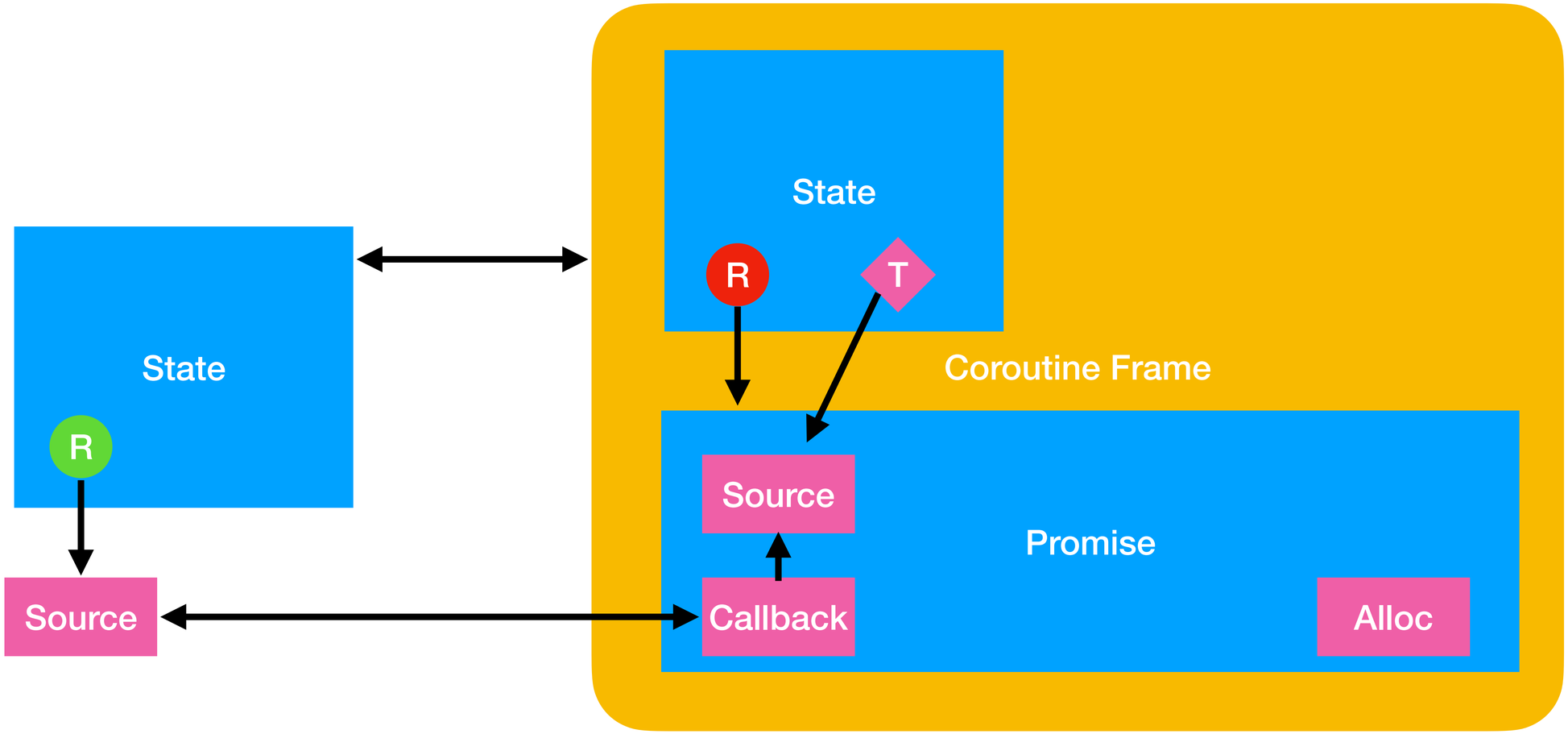
Task Forwards the Stop Token



Task Forwards the Stop Token



Task Forwards the Stop Token



Stop Token From Environment

```
struct stop_token_context {
    using stop_source_type = std::stop_source;
};
task<void, stop_token_context> stopping() {
    auto token = co_await read_env(get_stop_token);
    std::size_t count{};
    while (not token.stop_requested()) {
        ++count;
    }
}
```

Stop Token Support

- Cancellation is very important; thus, stop token support is important
- The stop token type comes from the receiver's environment:
  - task<…> type-erases the stop token type
  - Doing so requires de-/registering callbacks
- Use the receiver's stop token if the types match

User-Defined Query

```
constexpr struct get_v_t {
    template <typename E>
        requires requires(const get_v_t& self, const E& e) {
            e.query(self);
        }
    decltype(auto) operator()(const E& e) const {return e.query(*this);}
    constexpr bool query(const forwarding_query_t&)const noexcept
        { return true; }
} get_v{};
```

User-Defined Context

```
task<void> with_env() {
    decltype(auto) v = co_await read_env(get_v); // Error!
}
int main() { ex::sync_wait(with_env()); }
```

User-Defined Context

```
struct context {
    int value{};

    int query(get_v_t const&) const { return value; }
};

task<void, context> with_env() {
    decltype(auto) v = co_await read_env(get_v);
}
int main() { ex::sync_wait(with_env()); }
```


Bloomberg

Engineering

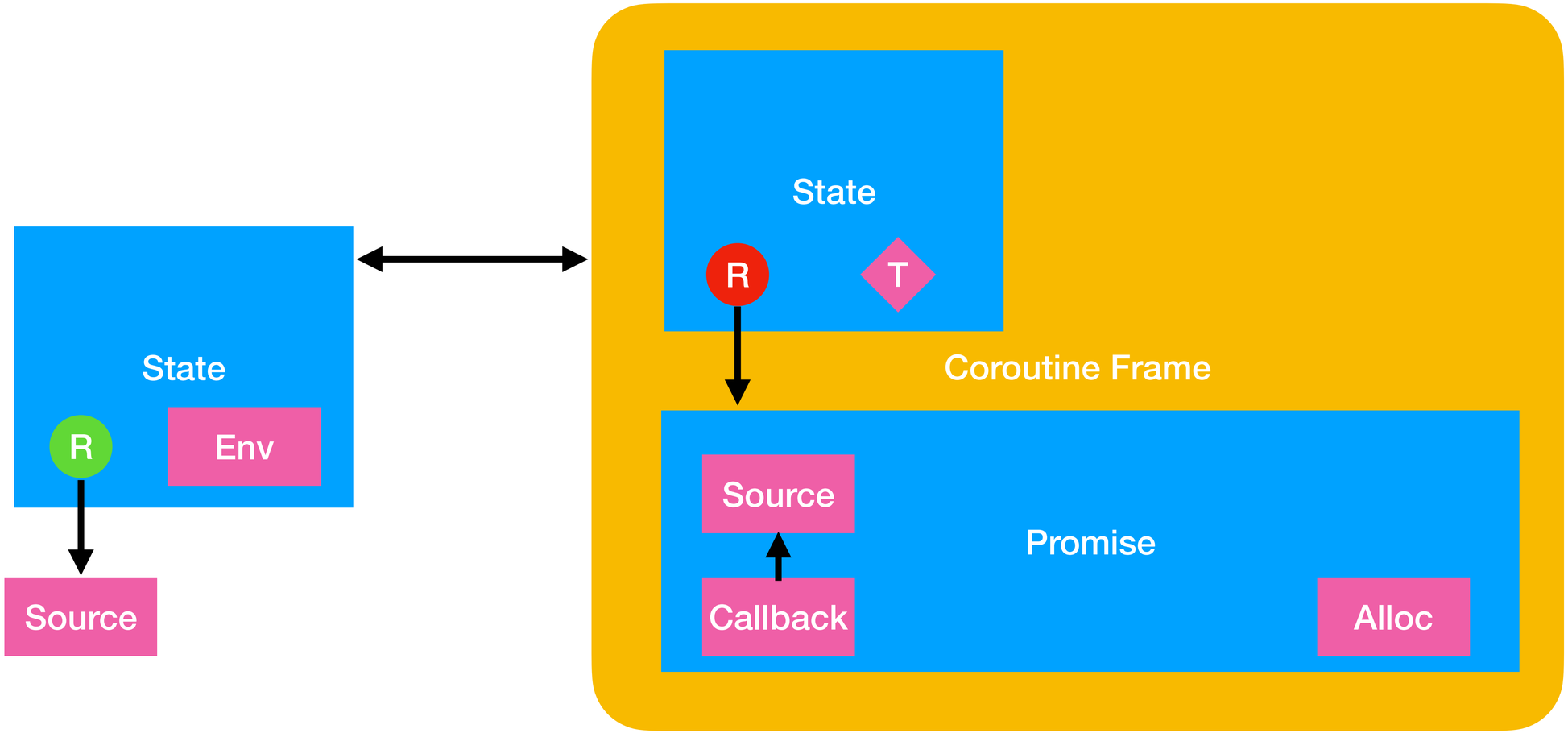
User-Defined Context

```
struct context {
    int value{};
    context(auto const& e): value(get_v(e)) {} // Error!
    int query(get_v_t const&) const { return value; }
};

task<void, context> with_env() {
    decltype(auto) v = co_await read_env(get_v);
}
int main() { ex::sync_wait(with_env()); }
```

User-Defined Context

```
struct context {
    int value{};
    context(auto const& e): value(get_v(e)) {}
    int query(get_v_t const&) const { return value; }
};

task<void, context> with_env() {
    decltype(auto) v = co_await read_env(get_v);
}
int main() { sync_wait(write_env(with_env(), make_env(get_v, 17))); }
```

Task Holds an Environment



User-Defined Context

```
struct fancy {
    struct env_base { virtual int get() const = 0; };
    template <typename Env> struct env_type : env_base {
        Env env;
        env_type(const Env& e) : env(e) {}
        int get() const override { return get_v(env); }
    };
    fancy(const env_base& own) : env(own) {} env_base const& env;
    int query(const get_v_t&) const { return this->env.get(); }
};
```

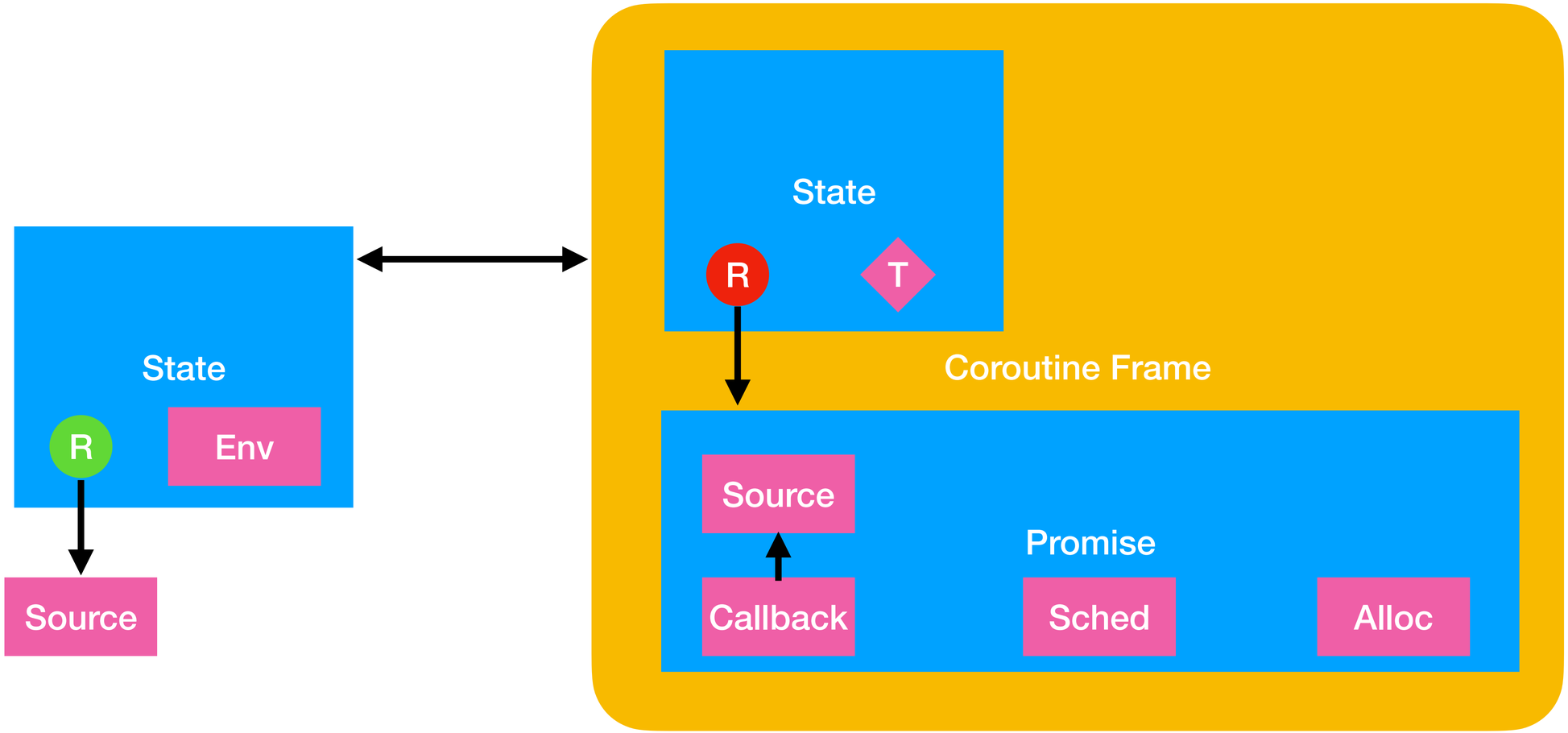
Where Does a Task Resume?

```
task<> work(auto sched) {
    // executing on orig_sched
    bool before = orig_sched == co_await read_env(get_scheduler);


    co_await (schedule(sched) | then([]{ /* executing on sched */ }));

    // executing on orig_sched or on sched?
}
```

Scheduler Affinity

```
task<> work(auto sched) {
    // executing on orig_sched
    bool before = orig_sched == co_await read_env(get_scheduler);


    co_await (schedule(sched) | then([]{ /* executing on sched */ }));

    // executing on orig_sched ~~or on sched~~?
}
```

Task Holds a Scheduler



Inhibiting Scheduler Affinity

```
task<> work(auto sched) {
    // executing on orig_sched
    bool before = orig_sched == co_await read_env(get_scheduler);
    co_await change_coroutine_scheduler(inline_scheduler{});

    co_await (schedule(sched) | then([]{ /* executing on sched */ }));

    // executing on orig_sched or on sched
}
```

Inhibiting Scheduler Affinity

```
struct inline_context { using scheduler_type = inline_scheduler; };
task<void, inline_context> work(auto sched) {
    // executing on orig_sched
    bool b = inline_scheduler == co_await read_env(get_scheduler);


    co_await (schedule(sched) | then([]{ /* executing on sched */ }));

    // executing on orig_sched or on sched
}
```

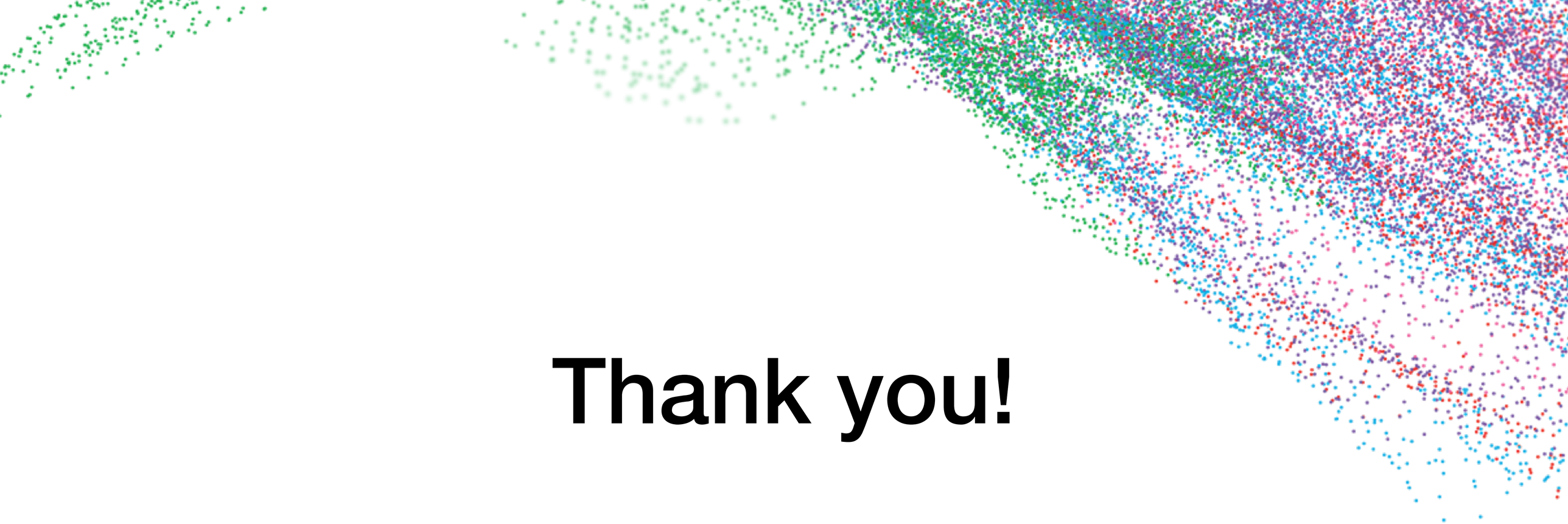

Resources

- Implementation: https://github.com/bemanproject/task
- Proposal: https://wg21.link/P3552

TechAtBloomberg.com

Bloomberg

Engineering


Thank you!