Managing External API's in Enterprise Systems

PETE MULDOON

Managing APIs in Enterprise Systems

Bloomberg

Engineering

CppCon 2022
Sept 13, 2022

Pete Muldoon
Senior Software Architect / Lead

TechAtBloomberg.com

Who Am I

- Starting using C++ professionally in 1991
- Professional Career
  - Systems Analyst & Architect
  - 21 years as a consultant
  - Bloomberg Ticker Plant Engineering Lead
- Talks focus on practical Software Engineering
  - Based in the real world
  - Take something away and be able to use it

Questions

#include <slide_numbers>

Where will we be going ?

- Talk will be about *using* APIs in applications
- Engineering decisions that were made
- Talk is based on a real world system not theory
- Early Design decisions were made prior to my participation but I will attempt to justify them.
  - Other paths to achieve these same aims could have been used
- Main focus will on handling change in a live production system

Enterprise Systems

Use many APIs

- Database
- Cache
- IPC
- Functionality

Providing

- Reference information
- Persistence
- Side effects propagated

APIs

What's out there ?

CppCon 2014 : "Hourglass Interfaces for C++ APIs“ - Stefanus DuToit
Meeting C++ 2015 : Creating intuitive APIs - Lars Knoll
Meeting C++ 2017 : API & ABI versioning - Mathieu Ropert
Meeting C++ 2017 : The most important API design principle - Marc Mutz
C++Now 2018 : “Modern C++ API Design” - Titus Winters
CppCon 2019 : Modern C++ API Design : two-day training course - Bob Steagall
C++Now 2019: “The C++ Reflection TS” - David Sankel
ACCU 2021 : The Business Value of a Good API - Bob Steagall
Meeting C++ online 2022 : Design of a C++ reflection API - Matúš Chochlík
C++ Weekly, Ep 295 2021 - API Design : Principle of Least Surprise
CppCon 2022 : Back to Basics : API Design - Jason Turner
Everywhere : < look at my snazzy API > - various
Everywhere : < How to use a specific API > - various

APIs

Good API design : Producer

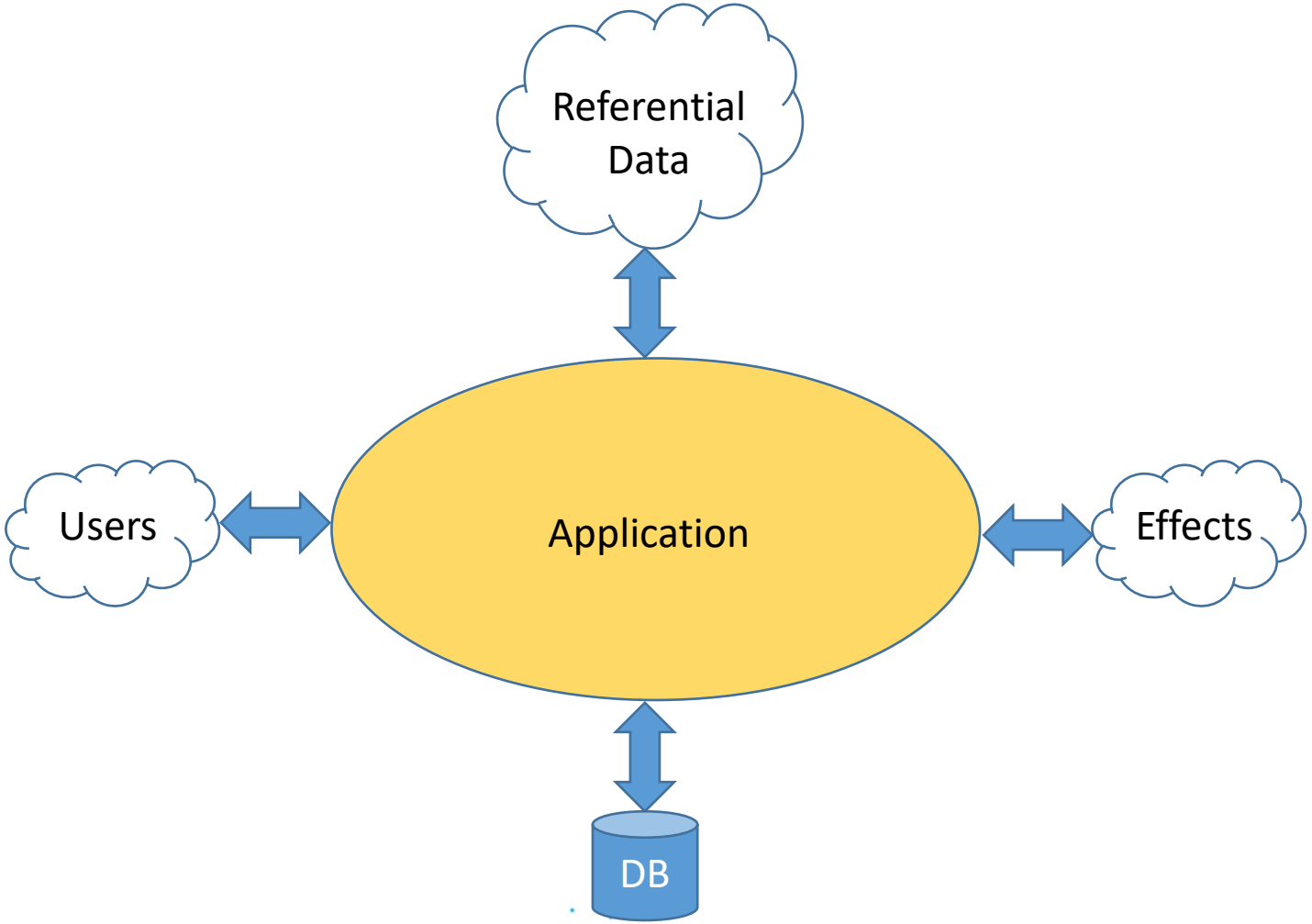
- General Use / flexible
- Many available modes of operation
- Testability
- Ergonomics

Good Use of APIs : Consumer

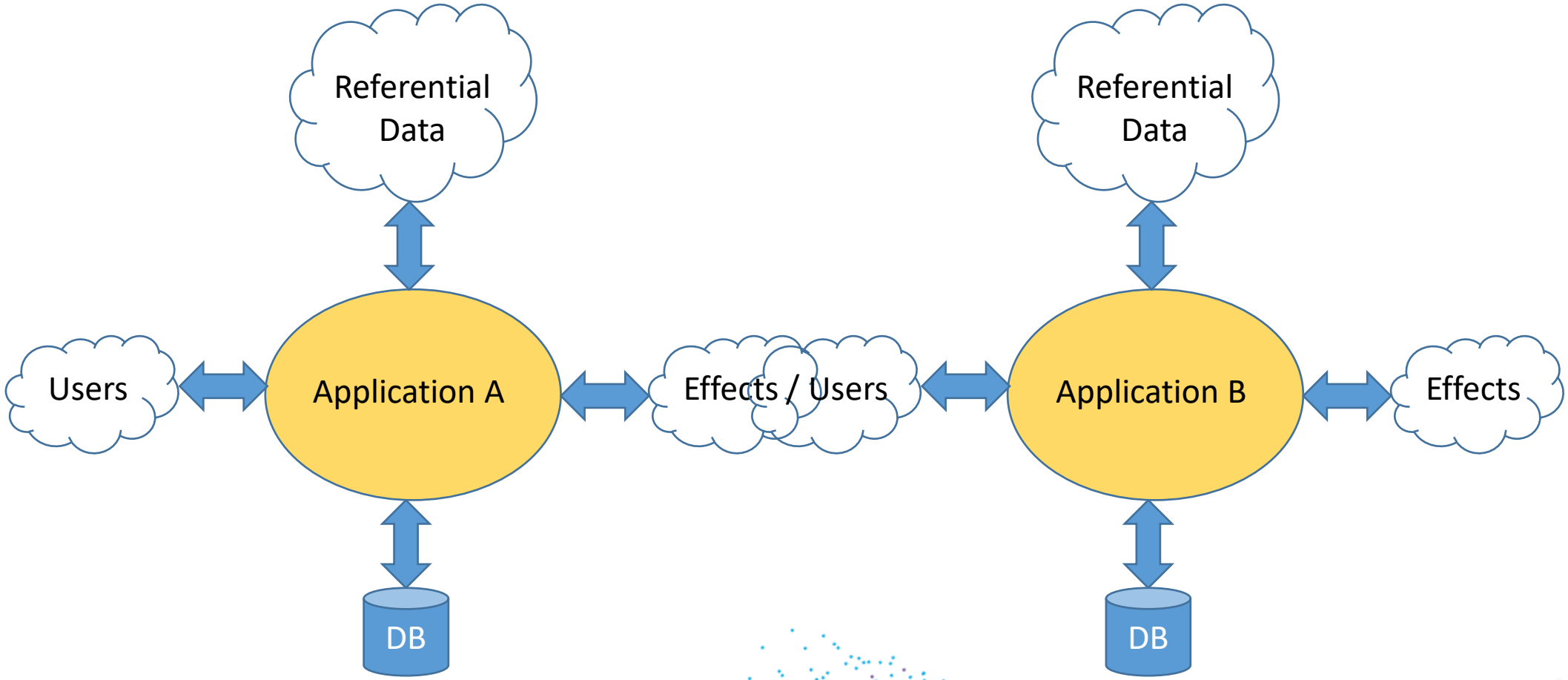
- Specific Use / Outcomes
- Constrained modes of operation
- Code efficiency / Bundling

Usually achieved via a Wrapper abstraction

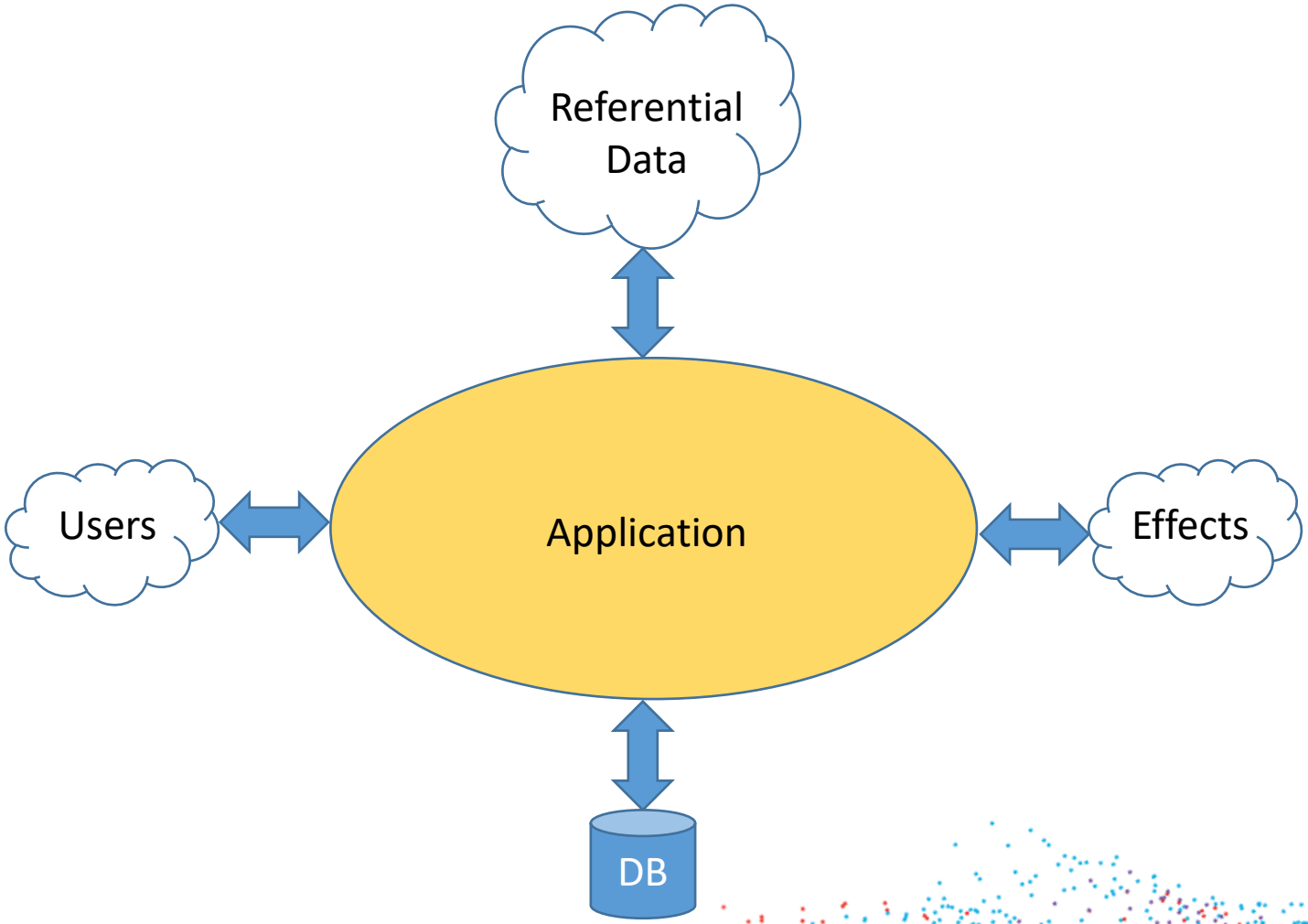
System



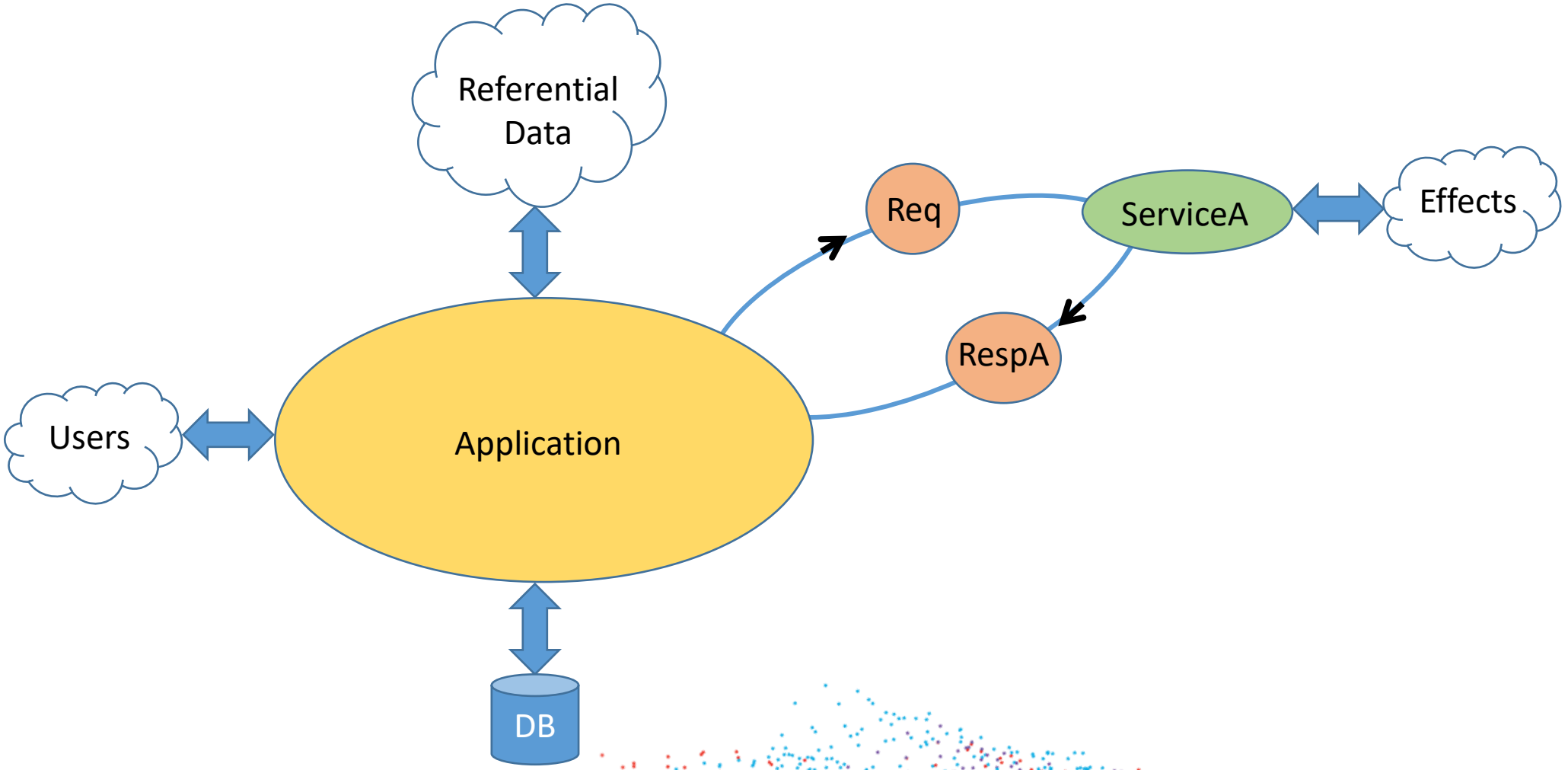
System Pipeline



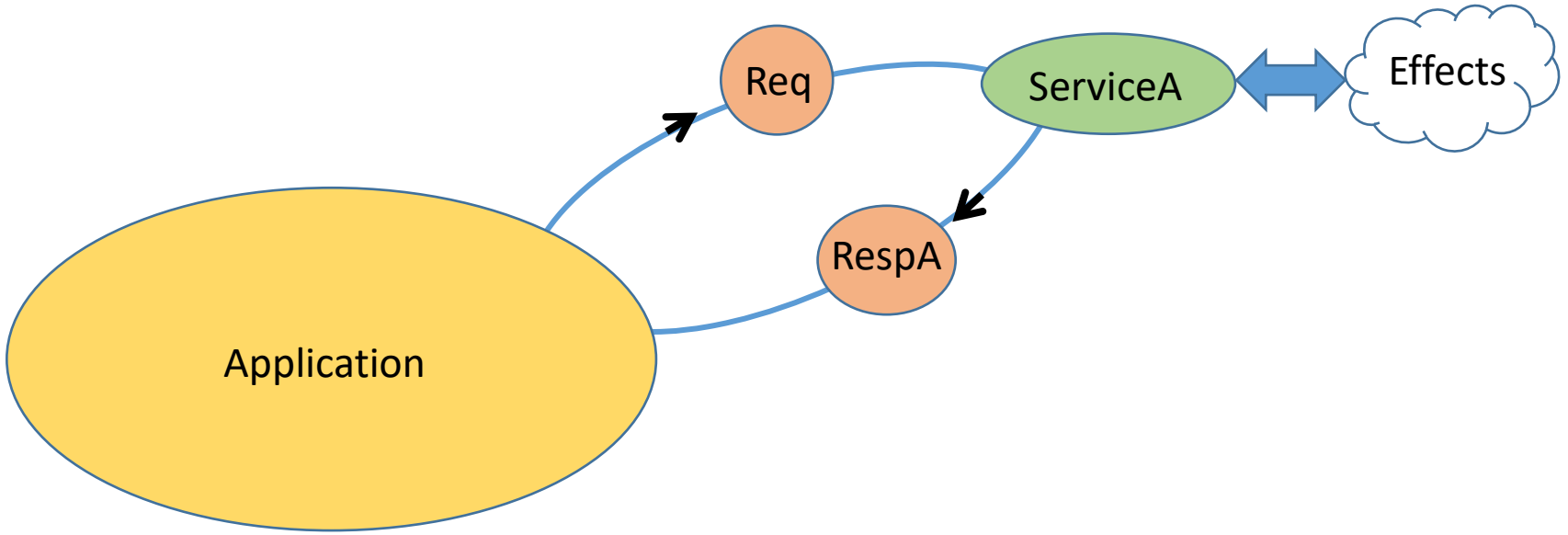
Full System



Full System


Referential Data
Req
ServiceA
Effects
RespA
Users
Application
DB

System Slice



External API

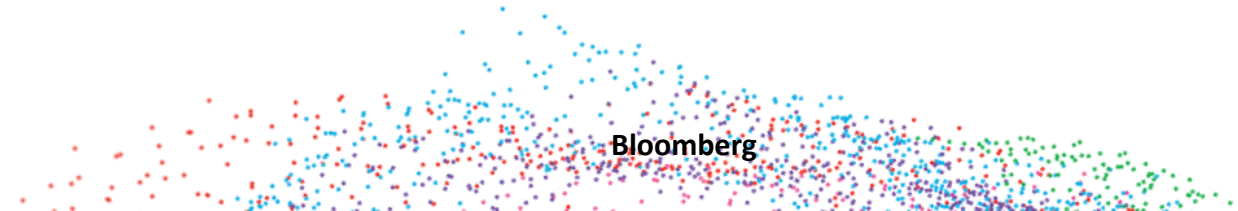
```
template<typename REQUEST, typename RESPONSE>
struct CommsApi {
  using Response = RESPONSE;
  …
  RESPONSE send(const REQUEST&);
  bool send(const REQUEST&, AsyncCallBack );
   …
};
```

Wrapper Design

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
};

template<typename REQUEST, typename RESPONSE>
struct Service {
  using Response = RESPONSE;

  RESPONSE send(const REQUEST&);

};
```

Wrapper Design

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
};

template<typename REQUEST, typename RESPONSE>
struct Service {
  using Response = RESPONSE;

 ResponseCarrier<RESPONSE> send(const REQUEST&);

};
```



Wrapper Design

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
};

template<typename REQUEST, typename RESPONSE>
struct Service {
  using Response = RESPONSE;

  std::future<ResponseCarrier<RESPONSE>> send(const REQUEST&);

};
```

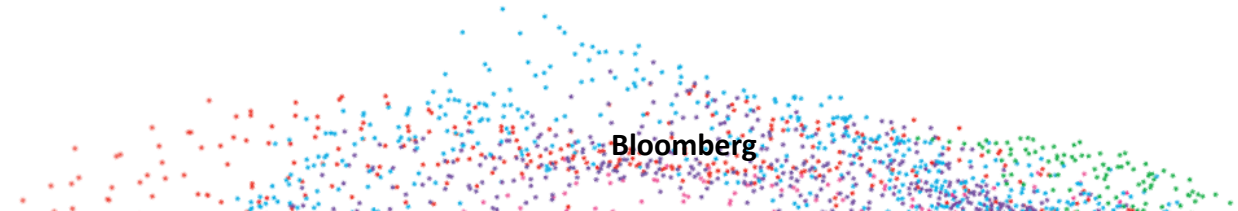
Application

```
// Service.h
using ServiceA = Service<Request, ResponseA>;


// main.cpp
ServiceA service_a{…};


// ErrorHandling.cpp
void check_response(const ResponseA& resp) {
  // if problem throw
}
```

```
// ActionXHandler.cpp
struct ActionXHandler {
  ServiceA& service_a;

  void execute(const ActionX& act)  {
    Request req;
    …
    auto result_future = service_a.send(req);
    auto result = result_future.get();
    check_response(*result.resp_ptr);
  }
};
```

Testing

```
// MockService.t.h
template <typename BASE>
struct MockService : public BASE {
  using Response = typename BASE::Response;
  using Carrier = ResponseCarrier<Response>;

  MOCK_CONST_METHOD1(onSend, std::future<Carrier>*(const Request& req));

  std::future<Carrier> send(const Request& req) {
    return std::move(*onSend(req));
  }
};
```



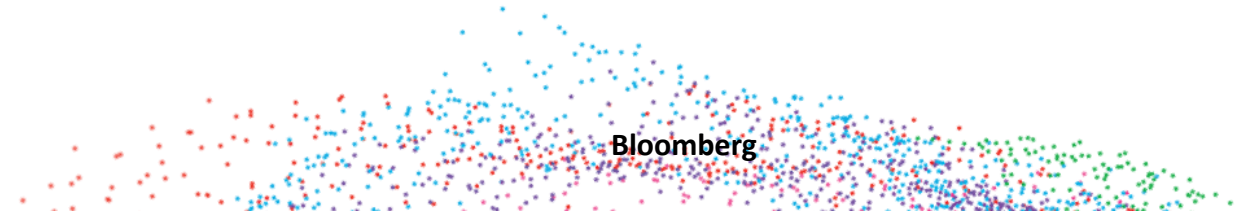
Wrapper Design

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
};

template<typename REQUEST, typename RESPONSE>
struct Service {
  using Response = RESPONSE;

  virtual std::future<ResponseCarrier<RESPONSE>> send(const REQUEST&);

};
```

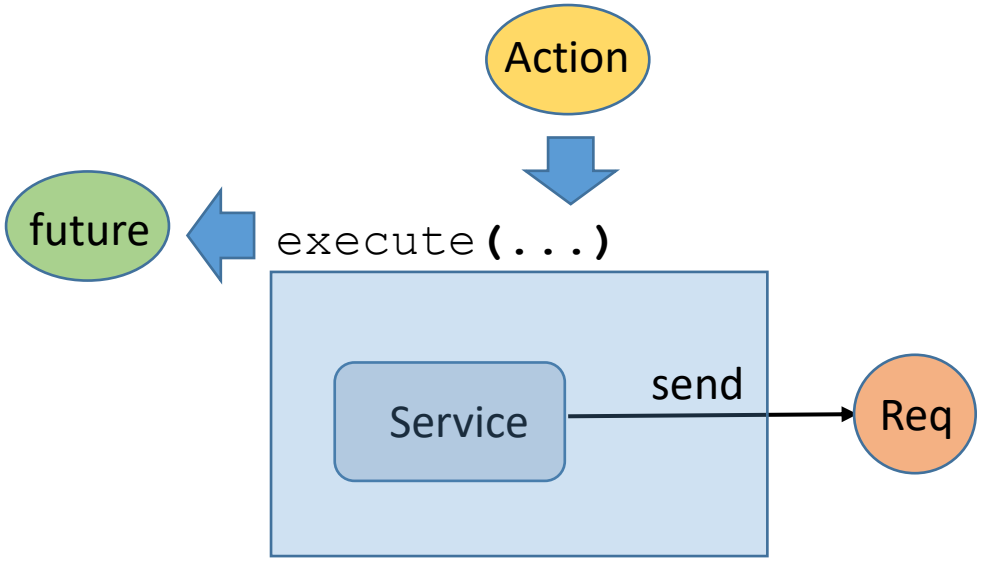
Testing

```
// ActionXHandler.h
struct ActionXHandler {
  ServiceA& service_a;

  auto execute(const ActionX& act) {
    Request req;
    // lots of logic to populate request
    …
    auto result = service_a.send(req).get();
    check_response(*result.resp_ptr);
    return result;
  }
};
```

```
// ActionXHandler.t.cpp
struct ActionXHandlerTests : public ::testing::Test {
  MockService<ServiceA> service_a;
  ActionXHandler handler{service_a};
  auto execute(const ActionX& act){ return handler.execute(act); }
 …
};
```



Testing



Testing

```
// ActionXHandler.t.cpp
...
TEST_F(ActionXHandlerTests, test_empty) {
  Request req;
  ActionX action {buildAction(...)};
  std::unique_ptr<std::future<ServiceA::Carrier>> resp{buildResp(false)};
  EXPECT_CALL(service_a, onSend(_)).WillOnce(DoAll(SaveArg<0>(&req), Return(resp.get())));
  EXPECT_THROW(execute(action), std::runtime_error);
}

TEST_F(ActionXHandlerTests, test_msg) {
  Request req;
  ActionX action {buildAction(2)};
  std::unique_ptr<std::future<ServiceA::Carrier>> resp{buildResp(true)};
  EXPECT_CALL(service_a, onSend(_)).WillOnce(DoAll(SaveArg<0>(&req), Return(resp.get())));
  EXPECT_NO_THROW(execute(action));
  EXPECT_EQ(req.qty(), 2);
  auto order = getOrder(...);
  EXPECT_EQ(order.qty(), 2);
  ...
}
```

Production

Time goes by ……

but

Change is in the air

Requirements Change

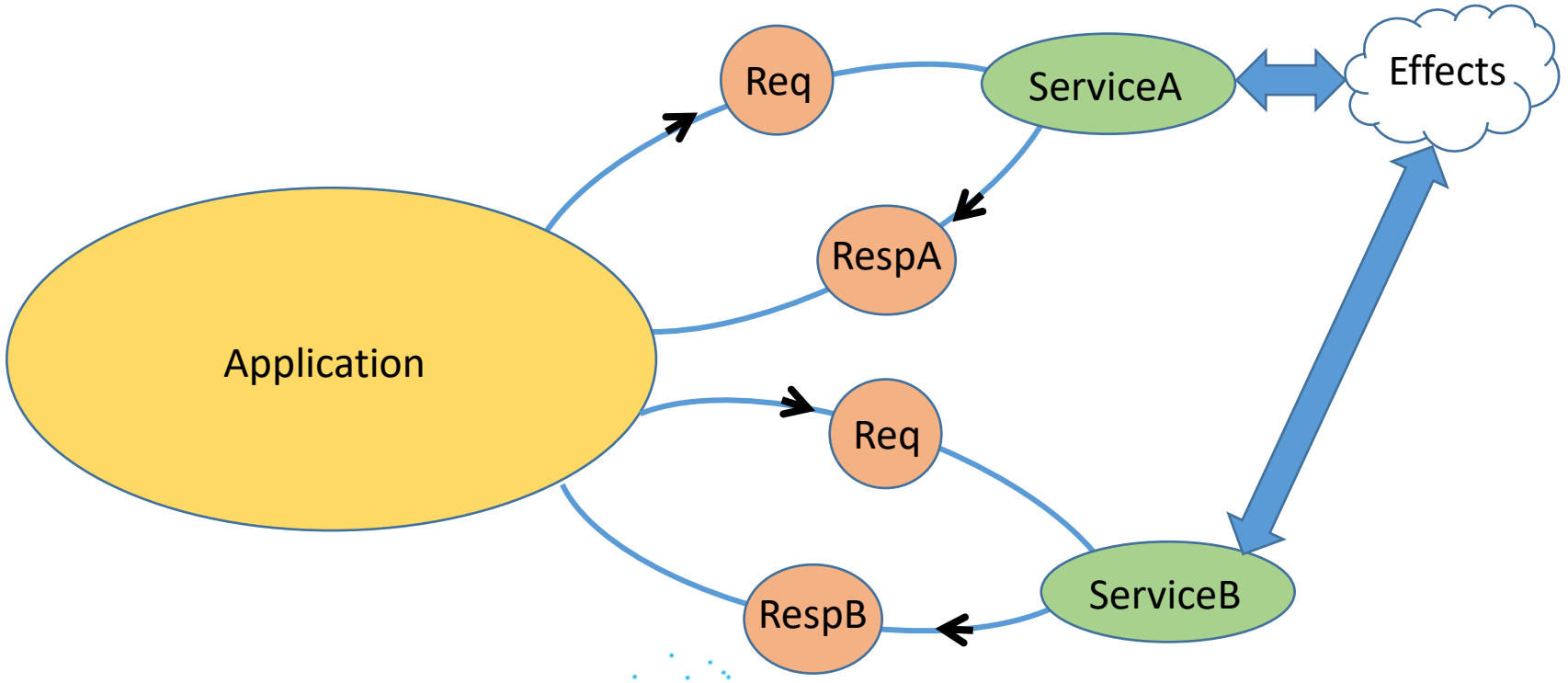
ServiceA to be replaced by ServiceB

- More direct
- Better throughput
- Better Latency
- Less points of failure

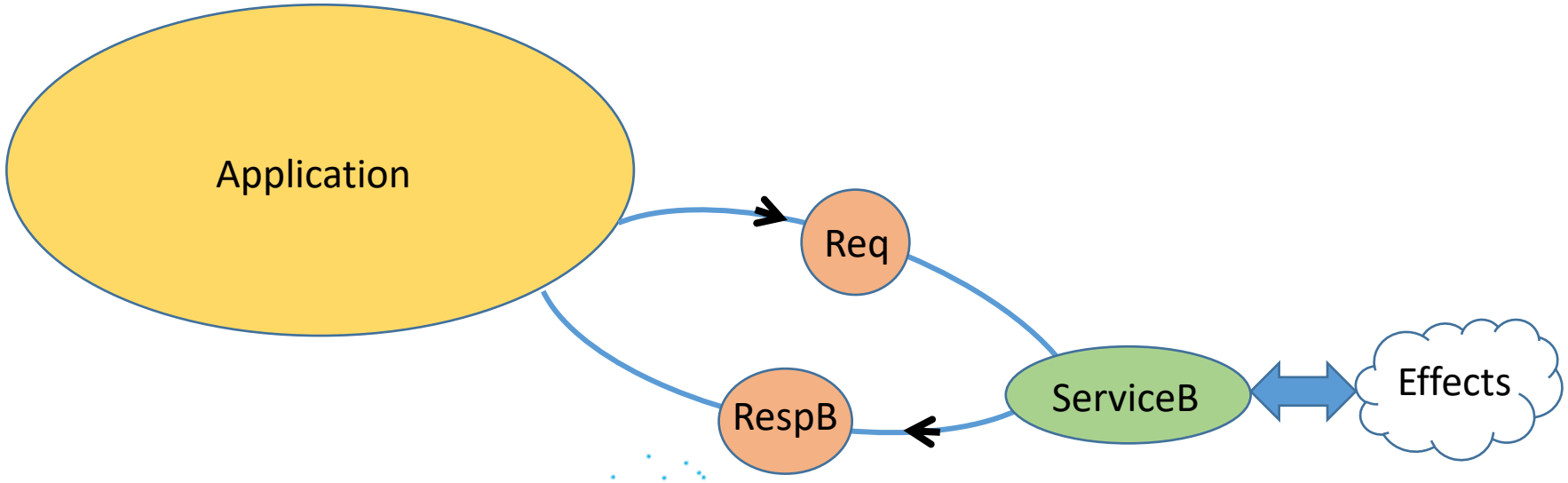
Note :

- New ServiceB is being built out – somewhat experimental

System Slice



System Slice



~~Straight Swap~~

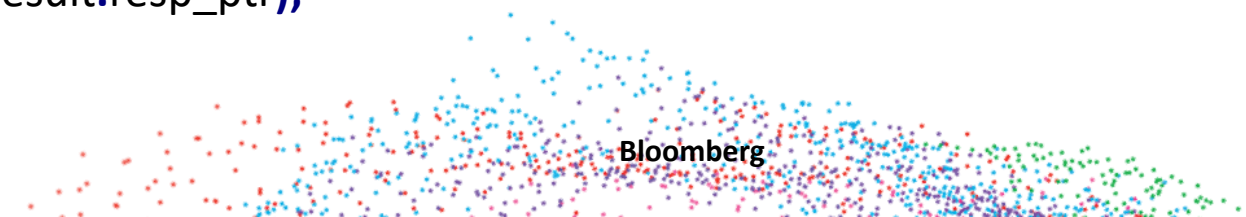
```
using ServiceB = Service<Request, ResponseB>;

ServiceB service;
```

```
void check_response(const ResponseB& resp) {
  // if problem throw
}

struct ActionXHandler {
   ServiceB& service_b;
   auto execute(const ActionX& act) {
     Request req{std::to_string(act.x)};
     // lots of logic
     auto result = service_b.send(req).get();
     check_response(*result.resp_ptr);
     return result;
   }
};
```

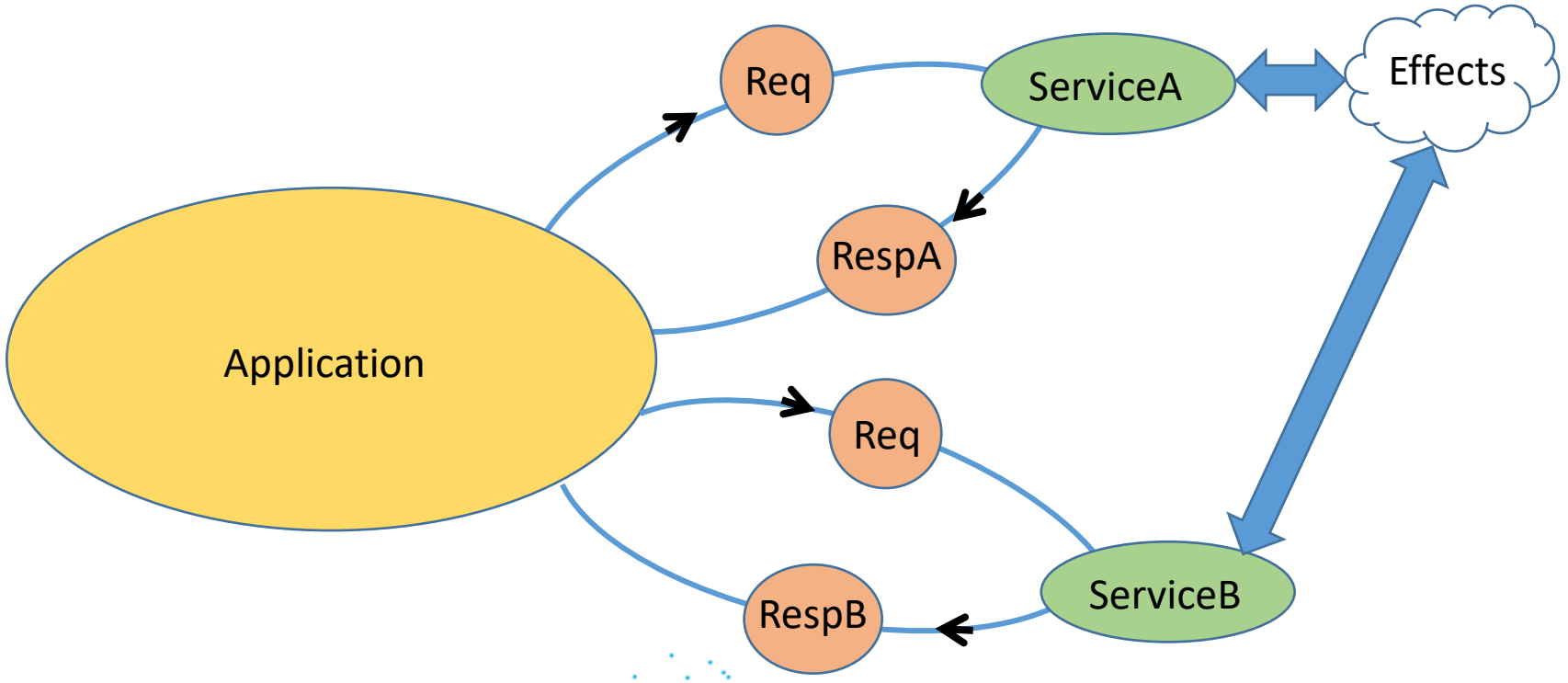
DONE ?

*** Approach is risky**

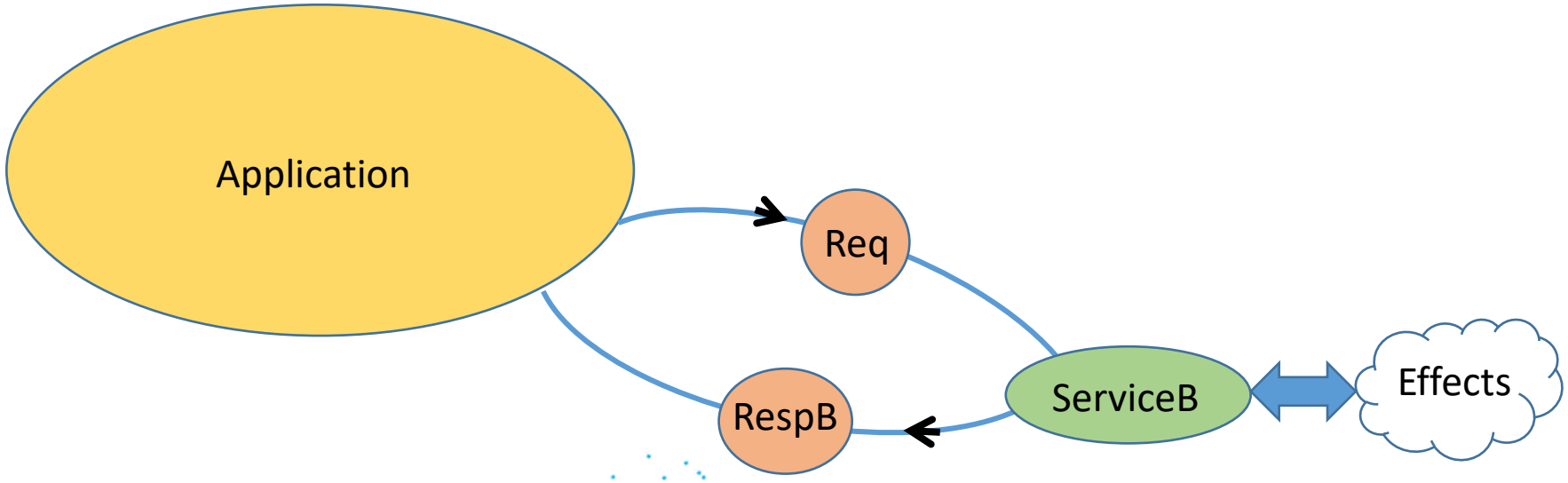
- New ServiceB *may* be unproven
- Interactions between your Application and ServiceB *are* unproven



Phased Approach



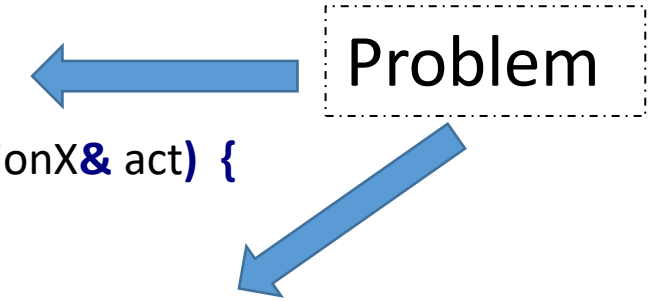
Phased Approach



~~Brute Force~~

```
// Service.h
using ServiceA = Service<Request, ResponseA>;
using ServiceB = Service<Request, ResponseB>;

// main.cpp
ServiceA service_a;
ServiceB service_b;

// ErrorHandling.cpp
void check_response(const ResponseA& resp) {
  // if problem throw
}

void check_response(const ResponseB& resp) {
  // if problem throw
}
```

```
// ActionXHandler.h

struct ActionXHandler {
  ServiceA& service_a;

  auto execute(const ActionX& act)  {
   Request req;
   ...
   if(route(req)) {
     auto result_future = service_a.send(req)
     auto result = result_future.get();
     check_response(*result.resp_ptr);
   } else {
     auto result_future = service_b.send(req)
     auto result = result_future.get();
     check_response(*result.resp_ptr);
   }
  }
}
```



Prioritize Migration Path

Requirements

- Localize meaningful changes
- Keep Global Usage/Calling Semantics unchanged
- Minimize throw-away work
- Final decommissioning simple

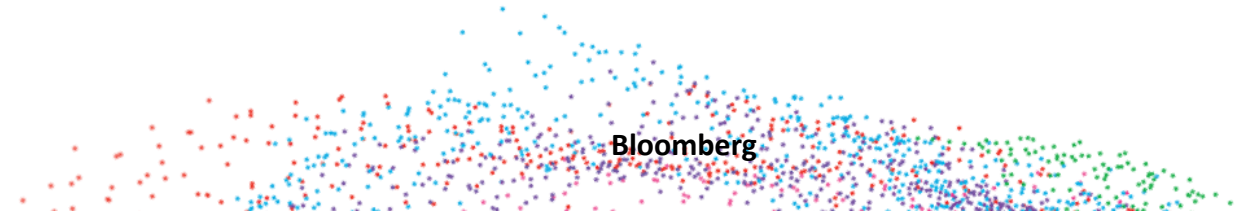
Watch out for Unit Testing

- Large amount of code
- Less Constrained than Production usage
- Mocks may have additional semantics

Prioritize Migration Path

```
// main.cpp
ServiceProxy proxy{…};

// ErrorHandling.cpp
void check_response(const ResponseProxy& resp) {
  // if problem throw
}
```

```
// ActionXHandler.h
struct ActionXHandler {
  ServiceProxy& proxy;
  auto execute(const ActionX& act) {
    Request req{std::to_string(act.x)};
    // lots of logic
    auto result = proxy.send(req).get();
    check_response(*result.resp_ptr);
    return result;
  }
};
```



Proxy Design

```
class ServiceProxy {
public:
 using VarRespFuture = std::variant<                              ,                              >;

 VarRespFuture send(const Request& req) {
   if(route(req))
     return VarRespFuture{service_a.send(req)};
   else
     return VarRespFuture{service_b.send(req)};
 }

private:
  ServiceA service_a;
  ServiceB service_b;
};
```

error: no member named 'get' in 'std::variant<std::future<ResponseCarrier<ResponseA>>, std::future<ResponseCarrier<ResponseB>>>'

Point of use

```
ServiceProxy service;

void check_response(const ResponseProxy& resp) {
  // if problem throw
}



struct ActionXHandler {
   ServiceProxy& proxy;
   auto execute(const ActionX& act) {
     Request req{std::to_string(act.x)};
     // lots of logic
     auto result_future = service.send(req)
     auto result = result_future.get();
     check_response(*result.resp_ptr);
  }
};
```

Problem

Future semantics

```
struct ResultFuture {
  using VarResp = std::variant<ServiceA::Carrier, ServiceB::Carrier>;
  VarRespFuture resp;

  VarResp get() {
    auto visitor = Overloaded{
      [&](std::future<ServiceA::Carrier>& resp_future) {
        return VarResp{resp_future.get()};
      },
      [&](std::future<ServiceB::Carrier>& resp_future) {
        return VarResp{resp_future.get()};
      }};
    return std::visit(visitor, resp);
  }
};
```

```
auto result = service.send(req).get();
```

Utility

```
// Utility Class
template<typename... Ts>
struct Overloaded : Ts... {
  using Ts::operator()...;
};
template<typename... Ts>
Overloaded(Ts&&...) -> Overloaded<std::decay_t<Ts>...>;
```

Future semantics

```
struct ResultFuture {
  using VarResp = std::variant<ServiceA::Carrier, ServiceB::Carrier>;
  VarRespFuture resp;

  VarResp get() {
    auto visitor = Overloaded{
       [&](std::future<ServiceA::Carrier>& resp_future) {
         return VarResp{resp_future.get()};
       },
       [&](std::future<ServiceB::Carrier>& resp_future) {
         return VarResp{resp_future.get()};
       }};
    return std::visit(visitor, resp);
  }
};
```

```
auto result = service.send(req).get();
```

Future semantics

```
struct ResultFuture {
  using VarResp = std::variant<ServiceA::Carrier, ServiceB::Carrier>;
  VarRespFuture resp;

  VarResp get() {
    return std::visit(
      [&](auto& resp_future) {
            return VarResp{resp_future.get()};
      }, resp);
  }
};
```

Proxy Design

```
class ServiceProxy {
public:
    using VarRespFuture = std::variant<std::future<ServiceA::Carrier>, std::future<ServiceB::Carrier>>;

ResultFuture send(const Request& req) {
    if(route(req))
      return ResultFuture{VarRespFuture{service_a.send(req)}};
    else
      return ResultFuture{VarRespFuture{service_b.send(req)}};
}

private:
  ServiceA service_a;
  ServiceB service_b;
};
```

error: no member named 'resp_ptr' in 'std::variant<ResponseCarrier<ResponseA>, ResponseCarrier<ResponseB>>'

Point of use

```
ServiceProxy service;

void check_response(const ResponseProxy& resp) {
  // if problem throw
}
```

```
struct ActionXHandler {
   ServiceProxy& proxy;
   auto execute(const ActionX& act) {
     Request req{std::to_string(act.x)};
     // lots of logic
     auto result_future = service.send(req)
     auto result = result_future.get();
     check_response(*result.resp_ptr);
   }
};
```

Problem

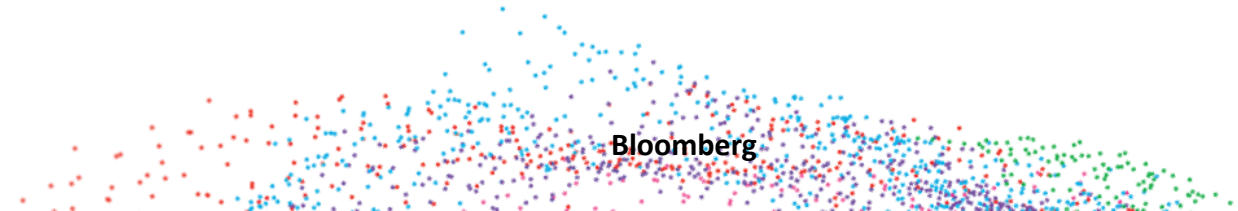
Result Semantics

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
};
```

Result Semantics

```
struct Result {
  VarResp& operator*() {
     return resp;
  }
  const VarResp& operator*() const {
     return resp;
  }
  VarResp resp;
  Result& resp_ptr = *this;
};
```

```
check_response(*result.resp_ptr);
```



Result Semantics

```
template<typename RESPONSE>
struct ResponseCarrier {
  std::unique_ptr<RESPONSE> resp_ptr;
  const RESPONSE& getResponse() const {
    return *resp_ptr;
  }
};
```

Point of use

```
struct Result {
  const VarResp& getResponse() const {
     return resp;
  }
  VarResp resp;
};
```

```
struct ActionXHandler {
  ServiceProxy& proxy;
  auto execute(const ActionX& act) {
    Request req;
    // lots of logic
    auto result_future = service.send(req)
    auto result = result_future.get();
    check_response(result.getResponse());
  }
};
```

Result Analysis

```
// ErrorHandling.cpp

void check_response(const ResponseA& ) {
  // if problem throw
}

void check_response(const ResponseB& ) {
  // if problem throw
}

void check_response(const ServiceProxy::VarResp& var_resp) {
     auto visitor = Overloaded{
        [&](const ServiceA::Carrier& resp) {
          check_response(resp.getResponse());
        },
        [&](const ServiceB::Carrier& resp) {
          check_response(resp.getResponse());
        }};
     return std::visit(visitor, var_resp);
}
```

Result Analysis

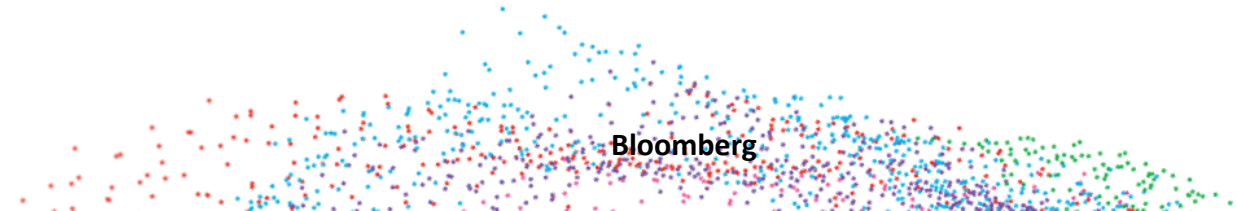
```
// ErrorHandling.cpp

void check_response(const ResponseA& ) {
  // if problem throw
}

void check_response(const ResponseB& ) {
  // if problem throw
}

void check_response(const ServiceProxy::VarResp& var_resp) {
    return std::visit(
        [&](const auto& resp) { check_response(resp.getResponse()); },
        var_resp);
}
```

Testing

```
// Mockers
template <typename BASE>
struct MockService : public BASE {
  using Response = typename BASE::Response;
  using Carrier = ResponseCarrier<Response>;
 MOCK_CONST_METHOD1(onSend, std::future<Carrier>*(const Request& req));

  std::future<ResponseCarrier<typename BASE::Response>> send(const Request& req) {
    return std::move(*onSend(req));
  }
};

struct MockServiceProxy : public ServiceProxy {
 MOCK_CONST_METHOD1(onSend, ResultFuture*(const Request& req));

 ResultFuture send(const Request& req) {
    return std::move(*onSend(req));
 }
};
```

Testing

```
// Tests
TEST(MockServiceTest, test_it) {
  MockServiceProxy proxy;
  Request req, o_req;
  std::unique_ptr<ServiceProxy::ResultFuture> resp{buildProxyResp()};
  EXPECT_CALL(proxy, onSend(_)).WillOnce(DoAll(SaveArg<0>(&req), Return(resp.get())));
  auto o_resp = proxy.send(o_req).get();
  EXPECT_THROW(check_response(o_resp.getResponse()), std::runtime_error);
}
```



Testing

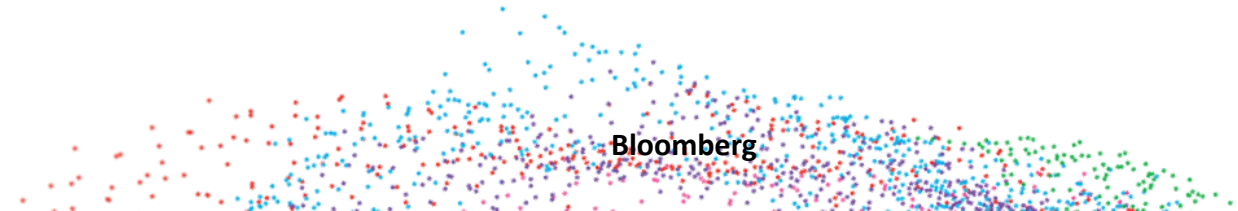
```
// Handler under test
struct ActionXHandler {
  ServiceProxy& proxy;
  auto execute(const ActionX& act) {
     Request req;
     // lots of logic
     auto result_future = proxy.send(req);
     auto result = result_future.get();
     check_response(result.getResponse());
};

// Test Fixture
struct ActionXHandlerTests : public ::testing::Test {
 MockServiceProxy proxy;
 ActionXHandler handler{proxy};
 auto execute(const ActionX& act){ return handler.execute(act); }
 ActionX buildAction(){ return ActionX{...};}
};
```

Testing

```
// Tests
TEST_F(ActionXHandlerTests, test_empty) {
  Request req;
  ActionX action {buildAction()};
  std::unique_ptr<ServiceProxy::ResultFuture> resp{buildProxyResp()};
  EXPECT_CALL(proxy, onSend(_)).WillOnce(DoAll(SaveArg<0>(&req), Return(resp.get())));
  EXPECT_THROW(execute(action), std::runtime_error);
}
```

Testing

```
// Tests
TEST_F(ActionXHandlerTests, test_msg) {
 Request req;
 ActionX action {buildAction(2)};
 std::unique_ptr<ServiceProxy::ResultFuture> resp{buildProxyResp()};
 EXPECT_CALL(proxy, onSend(_)).WillOnce(DoAll(SaveArg<0>(&req), Return(resp.get())));
 EXPECT_NO_THROW(execute(action));
 EXPECT_EQ(req.qty(), 2);
 auto order = getOrder(...);
 EXPECT_EQ(order.qty(), 2);
 ...
}
```



Decommission

```
class ServiceProxy {
public:
    using VarRespFuture = std::variant<std::future<ServiceB::Carrier>>;

ResultFuture send(const Request& req) {
   return ResultFuture{VarRespFuture{service_b.send(req)}};
}

private:
  ServiceB service_b;
};
```

Decommission

```
class ServiceProxy {
public:
    using ResultFuture = std::future<ServiceB::Carrier>;

ResultFuture send(const Request& req) {
   return service_b.send(req);
}

private:
  ServiceB service_b;
};
```

Reality Check

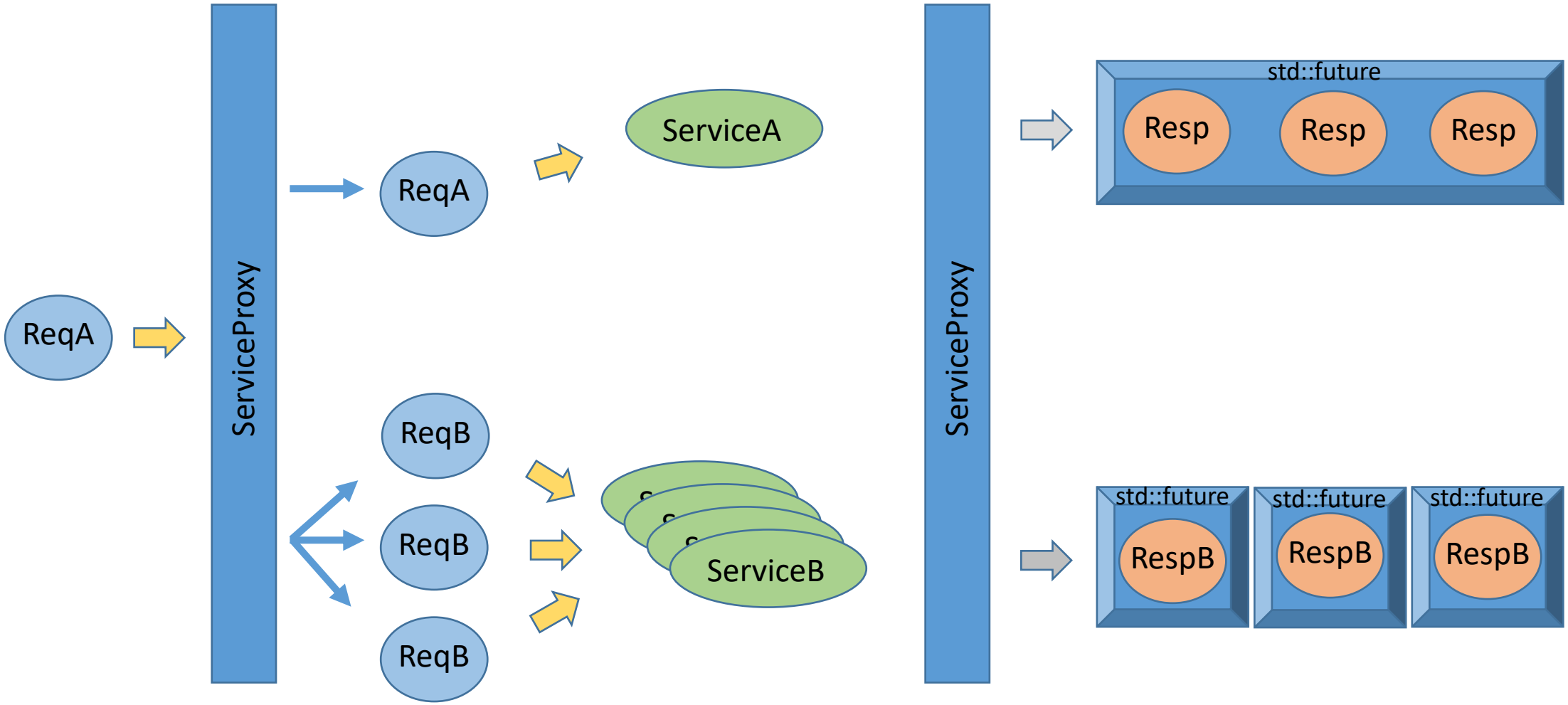
```
struct RequestA {
  std::vector<RequestB> payload;
};
```

```
struct RequestB {
  xxx payload;
};
```

```
struct ResponseA {
  std::future<std::vector<ResponseB>>
payload;
};
```

```
struct ResponseB {
  std::vector<std::future<ResponseB>> payload;
};
```

Reality Check



Change Priorities

API Use

- Channel good practices via wrapper with constraints
  - Move only data
  - Future/promise semantics
  - Layer of abstraction

Change Requirements

- Localize meaningful changes
- Keep Global usage/semantics constant
- Minimize throwaway work
- Final decommissioning simple

Endgame

Better delivery of change

Mitigate production risk

Godbolt listings

https://godbolt.org/z/814nds3Md - Rudimentary future returned
https://godbolt.org/z/GqxYnrWca - ResponseCarrier added
https://godbolt.org/z/7E94h95v4 - No member named get error
https://godbolt.org/z/MhTWhrqKj - No member named resp_ptr error
https://godbolt.org/z/r4nbWhPvP - Result class added
https://godbolt.org/z/WPPh7v4xq – ActionXHandler Tests with service
https://godbolt.org/z/6bKsGqs7G – ActionXHandler Tests with proxy
https://godbolt.org/z/r5bs93qs4 – Endgame with different requests and vectors
https://godbolt.org/z/GGafzzxrW - Endgame with templated lambda
https://godbolt.org/z/oKe3zqx6h - Decom simplification 1
https://godbolt.org/z/fr9eqTjsx - Decom simplification 2

Other Engineering talks :

Retiring The Singleton Pattern : Concrete suggestions what to use instead
Redesigning Legacy Systems : Keys to success

Questions ?

Bloomberg is hiring, come see us at the booth outside

Contact : pmuldoon1@Bloomberg.net