Take your microservices to the next level with gRPC

Mateusz Dymiński
Nokia

Whoami

Mateusz Dymiński

- Software Developer at Nokia
- 7+ exp with Java
- 3+ exp with Go
- One of the organizer GoWroc - Golang Wroclaw Meetup
- Github: github.com/mateuszdyminski
- Twitter: @m_dyminski
- LinkedIn: linkedin.com/in/mdyminski

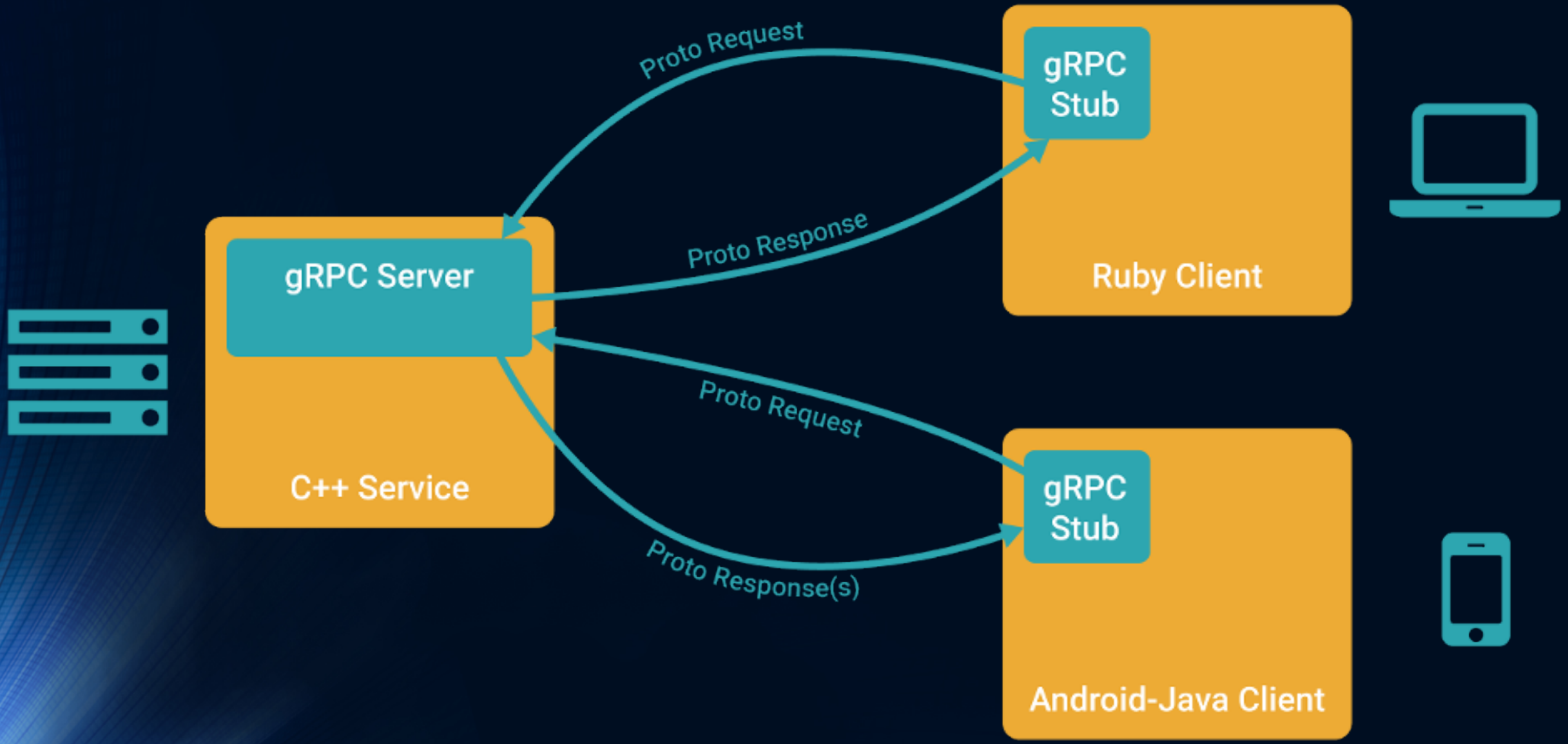
Agenda

- What's gRPC
- Problems in the microservices world
- How gRPC solves them
- Demo
- Summary
- Q&A

- How many of you have heard of gRPC?
- How many of you use microservices?
- How many of you use JSON to transfer your data between microservices

What is gRPC?

= RPC on steroids



Retrieved from: https://grpc.io/

RPC

- RPC – Remote Procedure Call
- Allows client and server to communicate
- Used in almost all distributed systems
- Solves problems: security, synchronization, and data flow handling etc
- Examples: NFS, D-Bus, JSON-RPC, SOAP, Apache Thrift, CORBA, Apache Avro, gRPC

gRPC

- RPC framework
- classic RPC and streaming RPC
- Multi-language: **Java**, **Go**, **C**, C++, Node.js, Python, Ruby, Objective-C, PHP and C#
- IDL: **Protocol Buffers 3, Flatbuffers**
- Transport: **HTTP2**
- Auth: **SSL/TLS**
- Open sourced: **github.com/grpc/**

gRPC – main usage scenarios

- Efficiently connecting polyglot services in microservices style architecture
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries
- Low latency, highly scalable, distributed systems.

Retrieved from: https://grpc.io/

gRPC – powered by

gRPC – Go tools

- Docker

- CockroachDB - Super stable distributed DB

- CoreOS/Etcd - Distributed consistent key-value store

- Google Cloud Bigtable - Sparse table storage

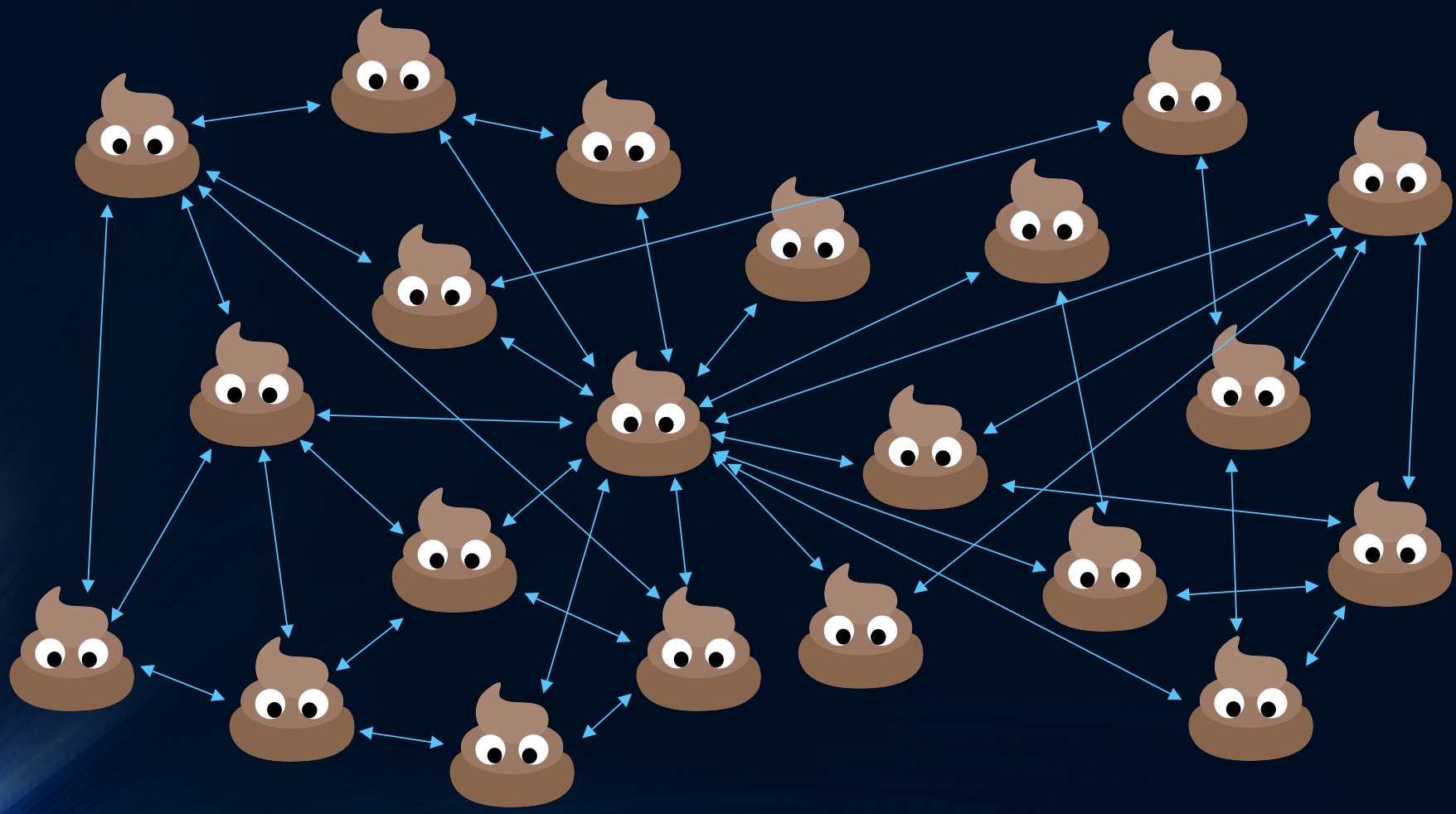
- YouTube/Vitess - Storage platform for scaling MySQL

gRPC – Java tools

- GCE pub/sub - Google Cloud Engine Pub/Sub client
- GCE Speech - Google Cloud Engine Speech client
- Netflix Ribbon - Inter Process Communication library
- Tensorflow - Scalable machine learning framework

Problems in Microservices world

Usually it looks like this:



You need to be this tall to use [micro] services:

here gRPC helps

- Distributed logging, tracing
- Know how to build, expose and maintain good APIs and contracts
- Ready to honor backward and forward compatibility, even if you're the same person consuming this service on the other side
- Provide Quality of service(QoS) - request deadline even if it's propagated to many different services
- Provide request propagations/cancellations - stop cascading failures
- Basic Monitoring, instrumentation, health checks
- Ready to isolate not just code, but whole build+test+package+promote for every service
- Can define upstream/downstream/compile-time/runtime dependencies clearly for each service
- Good unit testing skills and readiness to do more (as you add more microservices it gets harder to bring everything up, hence more unit/contract/api test driven and lesser e2e driven)
- Aware of [micro] service vs modules vs libraries, distributed monolith, coordinated releases, database-driven integration, etc
- Know infrastructure automation (you'll need more of it)
- Have working CI/CD infrastructure
- Have or ready to invest in development tooling, shared libraries, internal artifact registries, etc
- Have engineering methodologies and process-tools to split down features and develop/track/release them across multiple services (xp, pivotal, scrum, etc)

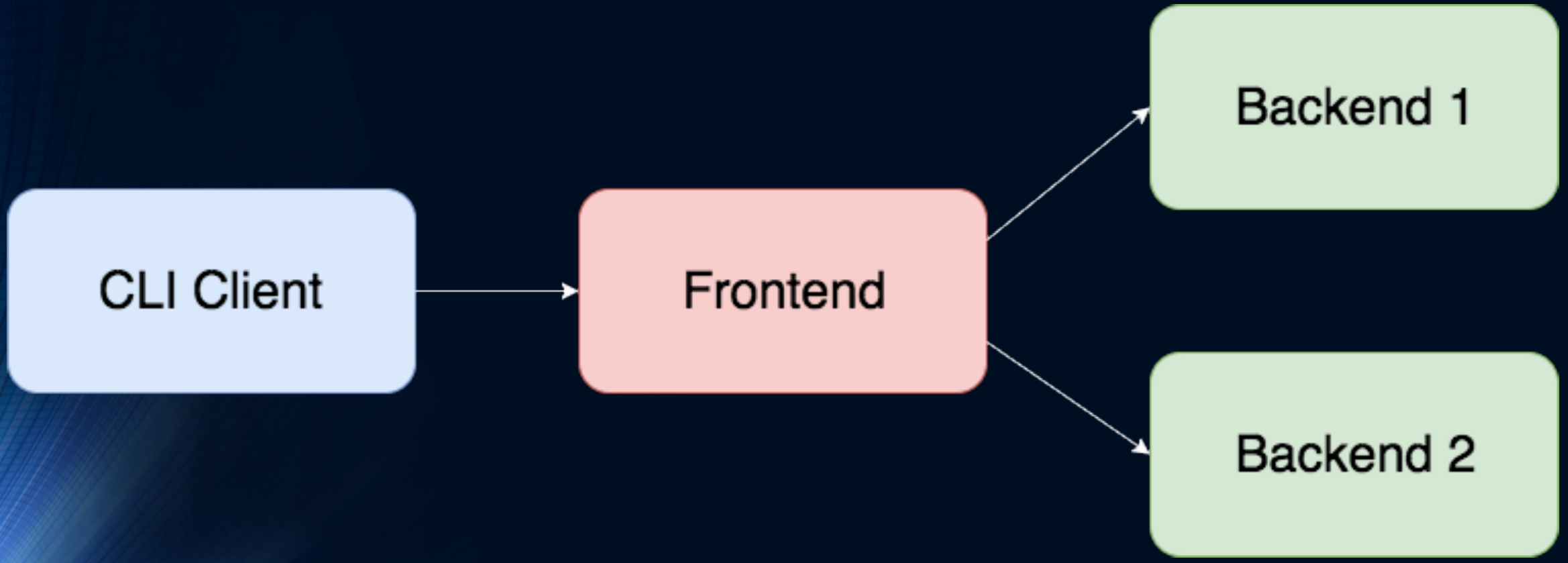
Retrieved from: HackerNews - https://news.ycombinator.com/item?id=12508655

Microservices - problems

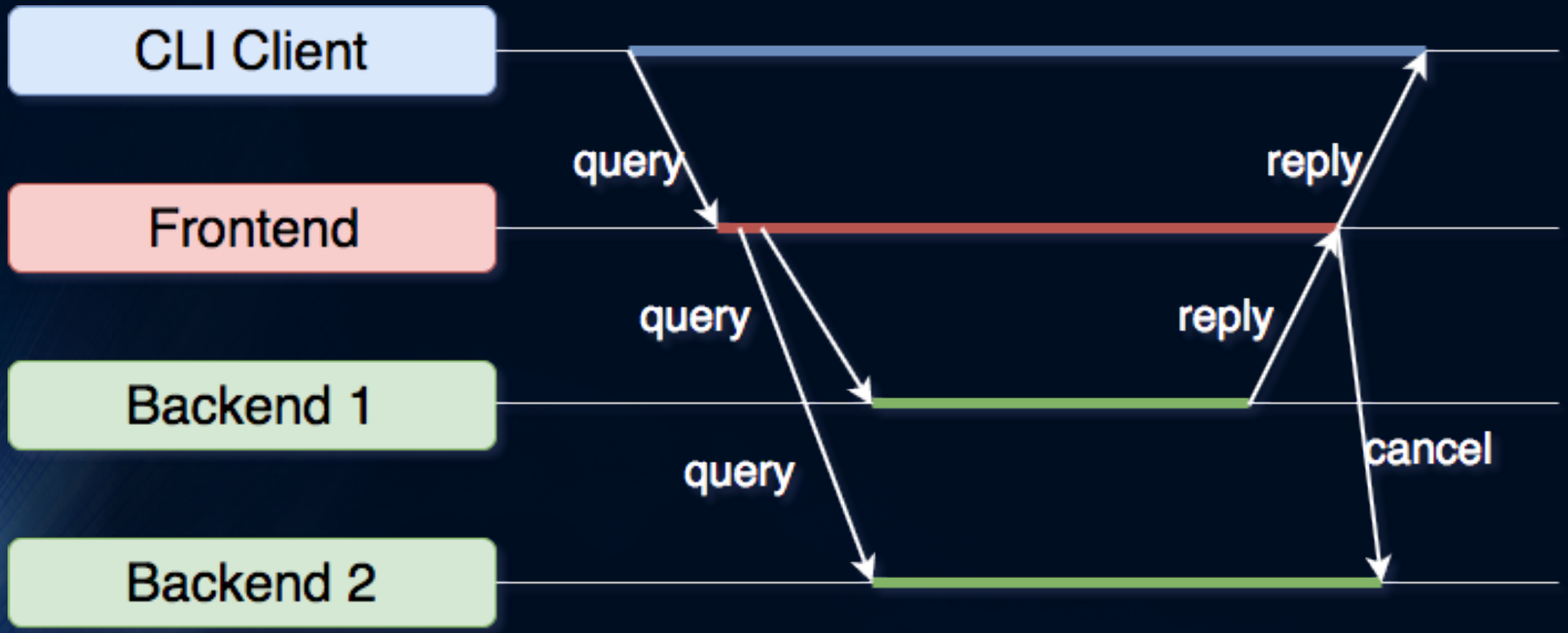
- Distributed tracing
- Build, expose and maintain good APIs and contracts
- Backward and forward compatibility
- Provide Quality of service(QoS) - deadline
- Provide request propagations/cancellations

Demo – Search Engine

Search Engine



Search Engine – timeline



Demo - Search Engine

- Frontend request tracing
- Backend request tracing
- Event logs

Protocol Buffers definition file

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "com.grpc.search";
option java_outer_classname = "SearchProto";
option objc_class_prefix = "GGL";
package search;

service Google {
  // Search returns a Search Engine result for the query.
  rpc Search(Request) returns (Result) {}
}
message Request {
  string query = 1;
}
message Result {
  string title = 1;
  string url = 2;
  string snippet = 3;
}
```

How to generate code – Go

Requirements:

- Protobuf github.com/google/protobuf
- Golang protoc wrapper:
  - go get github.com/golang/protobuf/protoc-gen-go

```
> protoc ./search.proto --go_out=plugins=grpc:../golang/search
```

How to generate code – Java

Requirements:

- Gradle/Maven
- protobuf-gradle-plugin/protobuf-maven-plugin

```
> gradle generateProto

> mvn protobuf:compile
```

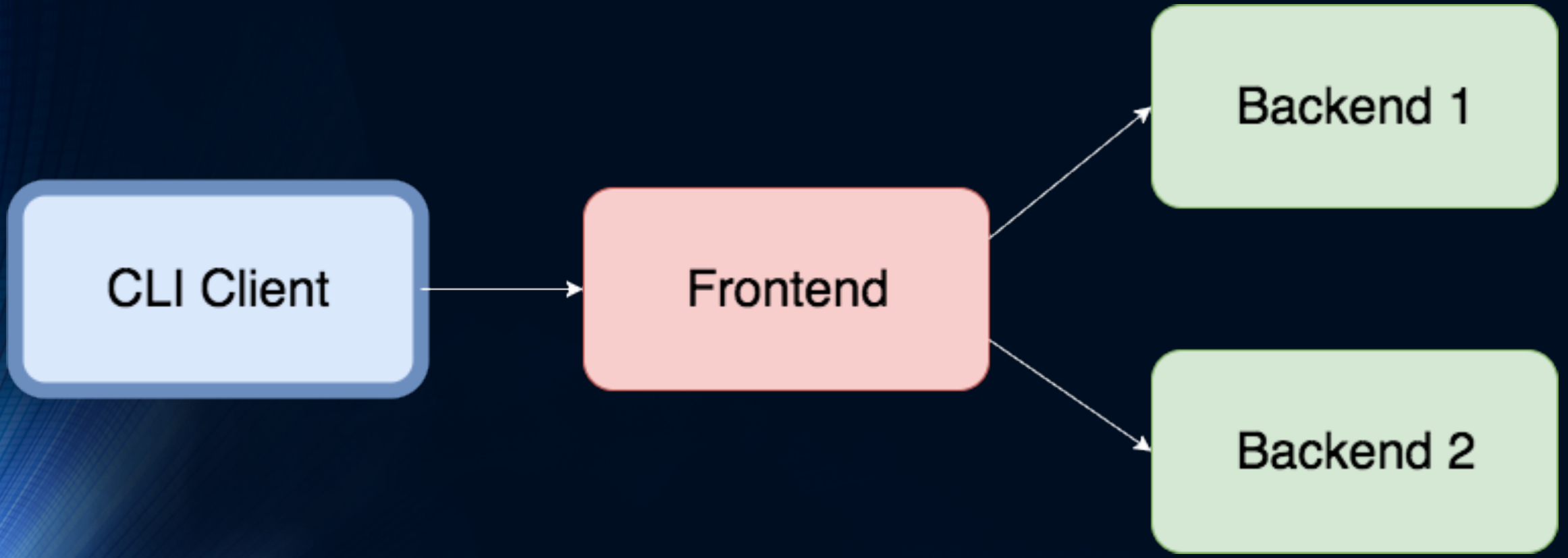
Generated code - Golang

```go
type SearchEngineServer interface {
  // Search returns a Search Engine result for the query.
  Search(context.Context, *Request) (*Result, error)
}

type SearchEngineClient interface {
  // Search returns a Search Engine result for the query.
  Search(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Result, error)
}

type Request struct {
  Query string `protobuf:"bytes,1,opt,name=query" json:"query,omitempty"`
}

type Result struct {
  Title   string `protobuf:"bytes,1,opt,name=title" json:"title,omitempty"`
  Url     string `protobuf:"bytes,2,opt,name=url" json:"url,omitempty"`
  Snippet string `protobuf:"bytes,3,opt,name=snippet" json:"snippet,omitempty"`
}
```

Generated code - Java

```
public  final class Request extends
    com.google.protobuf.GeneratedMessageV3 implements
    // @@protoc_insertion_point(message_implements:search.Request)
     RequestOrBuilder {

     public static final int QUERY_FIELD_NUMBER = 1;
     private volatile java.lang.Object query_;
     /**
      * <code>string query = 1;</code>
      */
     public java.lang.String getQuery() {
       java.lang.Object ref = query_;
       if (ref instanceof java.lang.String) {
          return (java.lang.String) ref;
       } else {
          com.google.protobuf.ByteString bs =
             (com.google.protobuf.ByteString) ref;
          java.lang.String s = bs.toStringUtf8();
          query_ = s;
          return s;
       }
     }
```

Source code – client



Source code – Java client

```
/**
 * BlockingClient is a client which blocks the execution until gets its response.
 */
public class BlockingClient implements SearchClient {

  private final ManagedChannel channel;
  private final SearchEngineGrpc.SearchEngineBlockingStub googleBlockingStub;

  public BlockingClient(String host, int port) {
    channel = ManagedChannelBuilder.forAddress(host, port)
        .usePlaintext(true)
        .build();

    googleBlockingStub = SearchEngineGrpc.newBlockingStub(channel);
  }

}
```

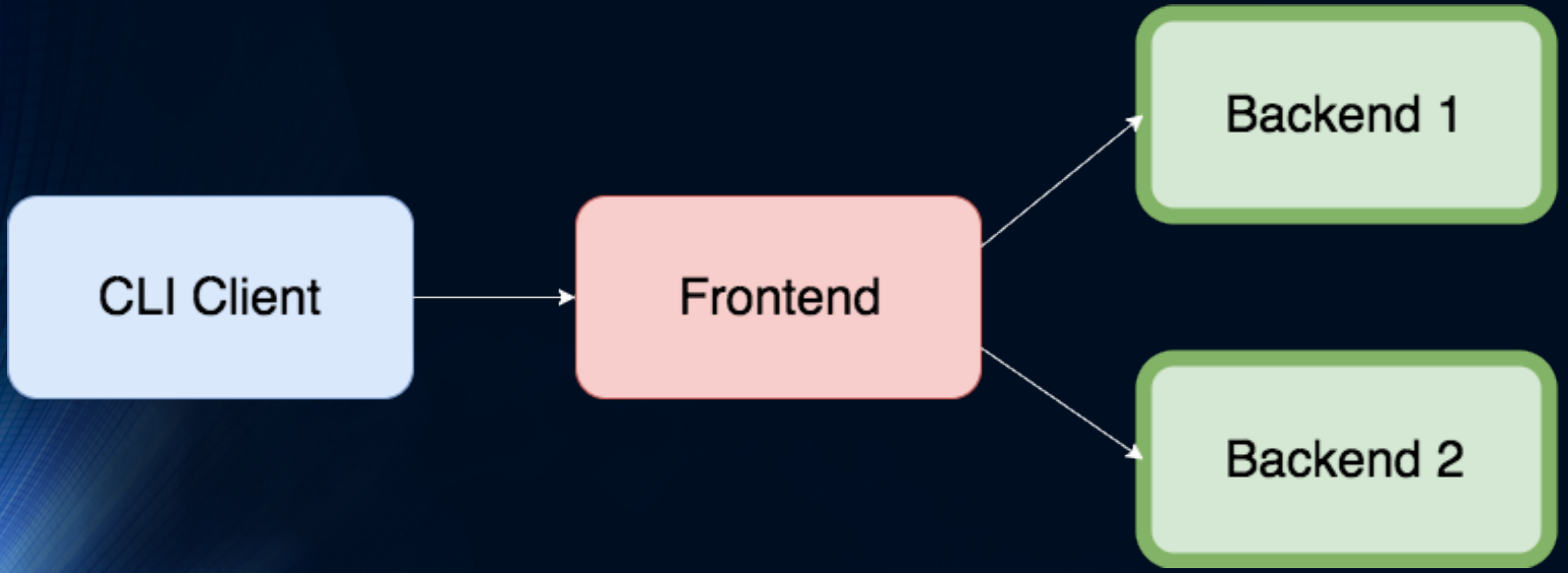
Source code – Java client - sync

```
/**
 * Search query in Search engine backend.
 */
@Override
public Result search(String query) {

  final Request request = Request.newBuilder().setQuery(query).build();

  return Try
        .ofFailable(() -> googleBlockingStub.search(request))
        .onSuccess(r -> logger.info("Search result: " + r))
        .onFailure(e -> logger.log(Level.SEVERE, "RPC failed: {0}", e))
        .getUnchecked();
}
```

Source code – Java client - async

```java
/**
 * Search searches query in async way in Search engine backend.
 */
@Override
public Result search(String query) {

  Request request = Request.newBuilder().setQuery(query).build();
  ListenableFuture<Result> resultFuture = googleFutureClient.search(request);

  return Try.ofFailable(() -> resultFuture.get(1000, TimeUnit.MILLISECONDS))
        .onSuccess(r -> logger.info("Search result: " + r))
        .onFailure(e -> logger.log(Level.SEVERE, "RPC failed: {0}", e))
        .getUnchecked();
}
```

Source code - backend



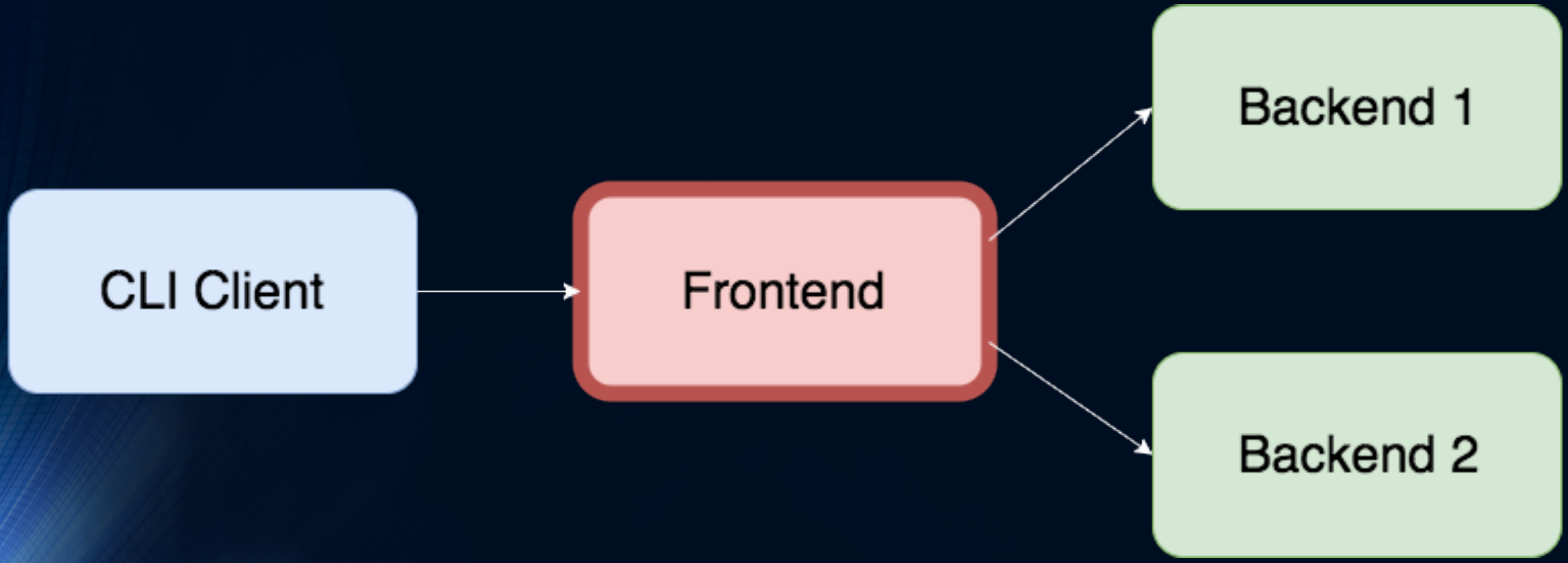
Source code – Go backend

```
func main() {
    flag.Parse()
    rand.Seed(time.Now().UnixNano())
    go http.ListenAndServe(fmt.Sprintf(":%d", 36661+*index), nil)   // HTTP debugging
    lis, err := net.Listen("tcp", fmt.Sprintf(":%d", 36061+*index)) // RPC port
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }


    g := grpc.NewServer()
    pb.RegisterSearchEngineServer(g, new(server))
    g.Serve(lis)
}
```

Source code – Go backend

```
// Search sleeps for a random interval then returns a string
func (s *server) Search(ctx context.Context, req *pb.Request) (*pb.Result, error) {
    d := randomDuration(100 * time.Millisecond)
    logSleep(ctx, d)
    select {
    case <-time.After(d):
        return &pb.Result{
            Title: fmt.Sprintf("result for [%s] from backend %d", req.Query, *index),
        }, nil
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}
```

Source code - frontend



Source code – Go frontend

```
// Search issues Search RPCs in parallel to the backends and returns first result.
func (s *server) Search(ctx context.Context, req *pb.Request) (*pb.Result, error) {
  c := make(chan result, len(s.backends))
  for _, b := range s.backends {
    go func(backend pb.SearchEngineClient) {
      res, err := backend.Search(ctx, req)
      c <- result{res, err}
    }(b)
  }
  first := <-c
  return first.res, first.err
}

type result struct {
  res *pb.Result
  err error
}
```

Source code – Go frontend

```
func main() {
  go http.ListenAndServe(":36660", nil)   // HTTP debugging
  lis, err := net.Listen("tcp", ":36060") // RPC port
  if err != nil {
    log.Fatalf("failed to listen: %v", err)
  }
  s := new(server)
  for _, addr := range strings.Split(*backends, ",") {
    conn, err := grpc.Dial(addr, grpc.WithInsecure())
    if err != nil {
      log.Fatalf("fail to dial: %v", err)
    }
    client := pb.NewSearchEngineClient(conn)
    s.backends = append(s.backends, client)
  }

  g := grpc.NewServer()
  pb.RegisterSearchEngineServer(g, s)
  g.Serve(lis)
}
```

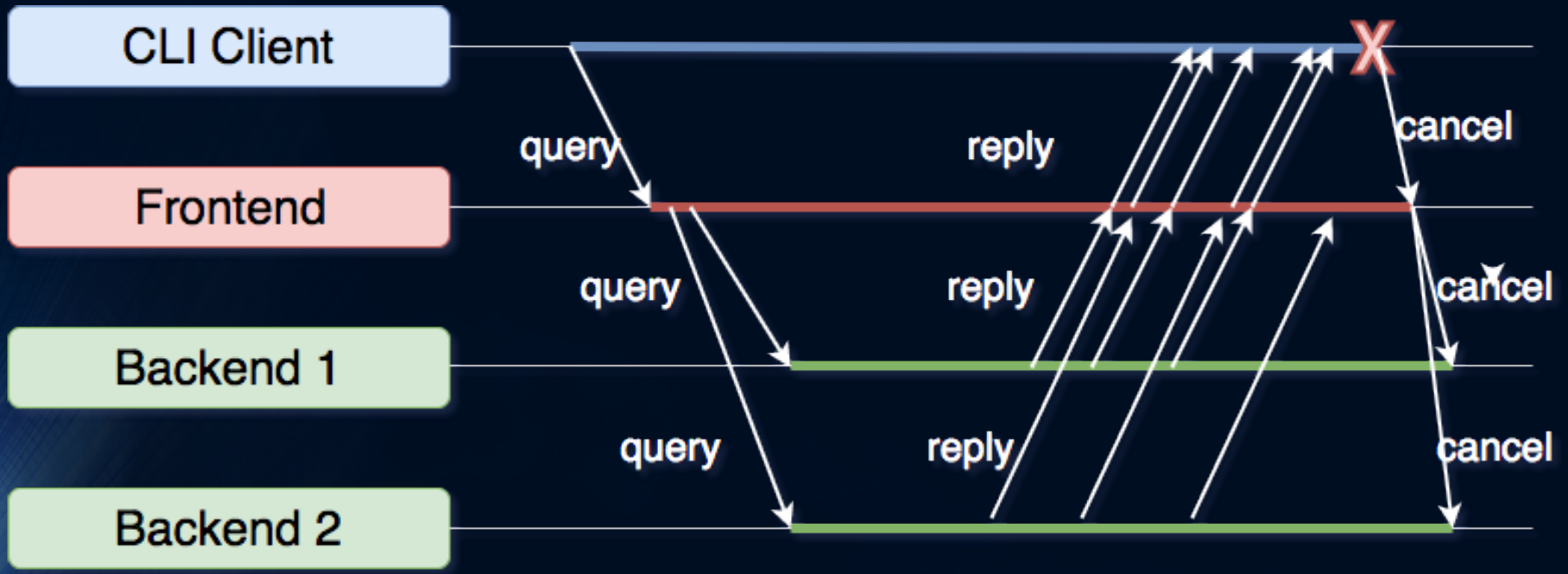
What is better than a demo?

Another demo!

Demo – Streaming Search Engine

- Debugging active stream
- Request cancelation

Streaming Search Engine

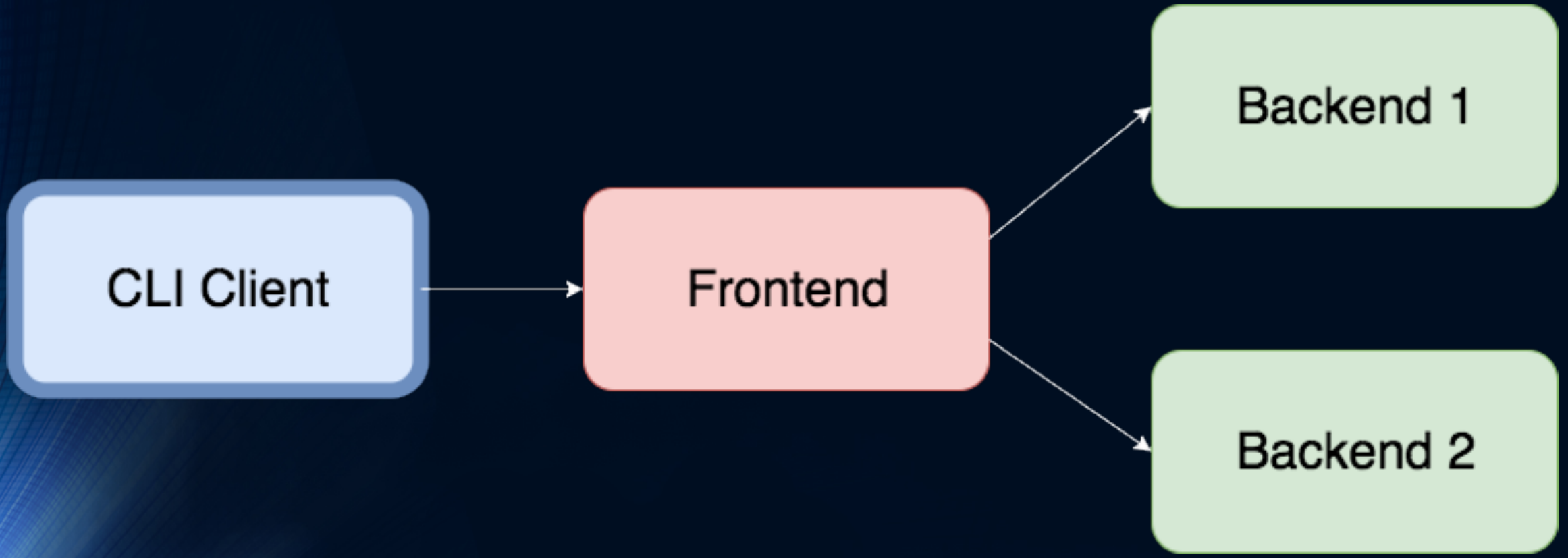


Protocol Buffers definition file

New method >

```
syntax = "proto3";
option java_multiple_files = true;
option java_package = "com.grpc.search";
option java_outer_classname = "SearchProto";
option objc_class_prefix = "GGL";
package search;

service Google {
 // Search returns a Search Engine result for the query.
 rpc Search(Request) returns (Result) {}

 // Watch returns a stream of Search Engine results for the query.
 rpc Watch(Request) returns (stream Result) {}
}
message Request {
 string query = 1;
}
message Result {
 string title = 1;
 string url = 2;
 string snippet = 3;
}
```

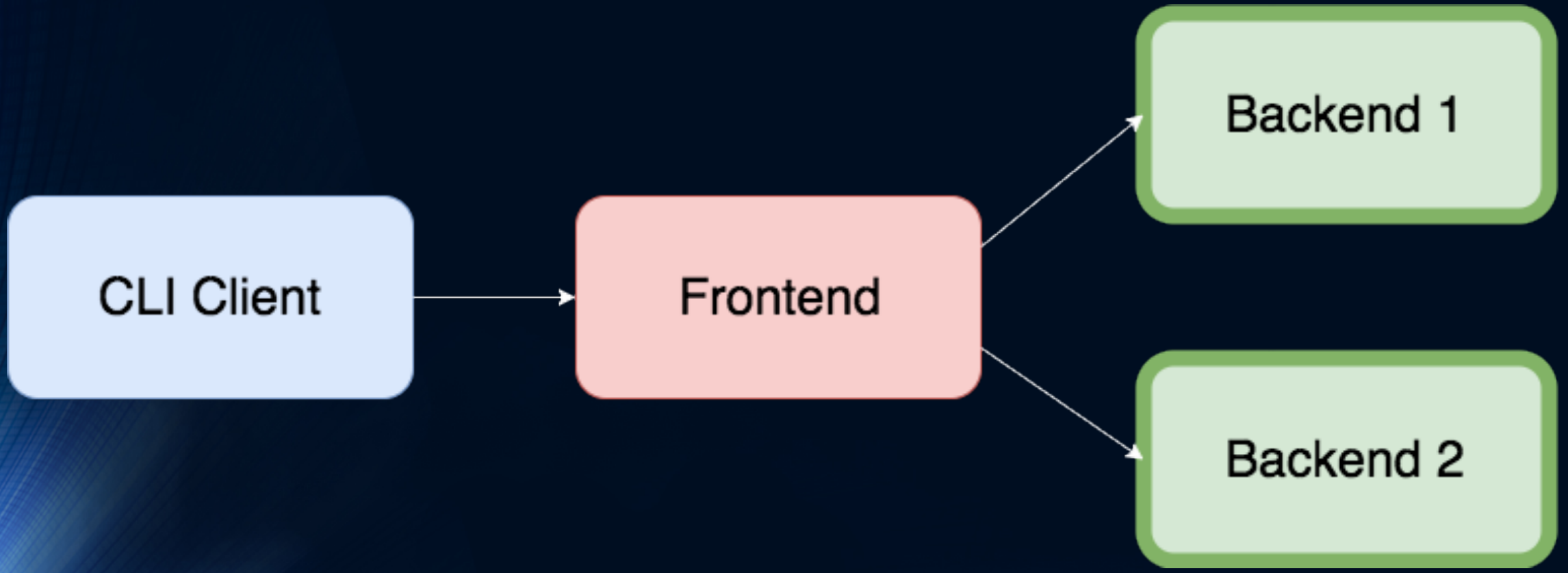
Source code - client



Source code – Java client - streaming

```
public void watch(String query) {
  final Request request = Request.newBuilder().setQuery(query).build();
  final CountDownLatch latch = new CountDownLatch(1); // we expect only 1 result

  StreamObserver<Result> stream = new StreamObserver<Result>() {
    public void onNext(Result value) {
      logger.info("Search result: " + value.getTitle());
    }

    public void onError(Throwable t) {
      logger.severe(("Error while watching for results! " + t.getMessage()));
      latch.countDown();
    }

    public void onCompleted() {
      logger.info("Watch done!");
      latch.countDown();
    }
  };

  googleStub.watch(request, stream);
  Uninterruptibles.awaitUninterruptibly(latch, 100, TimeUnit.SECONDS);
}
```

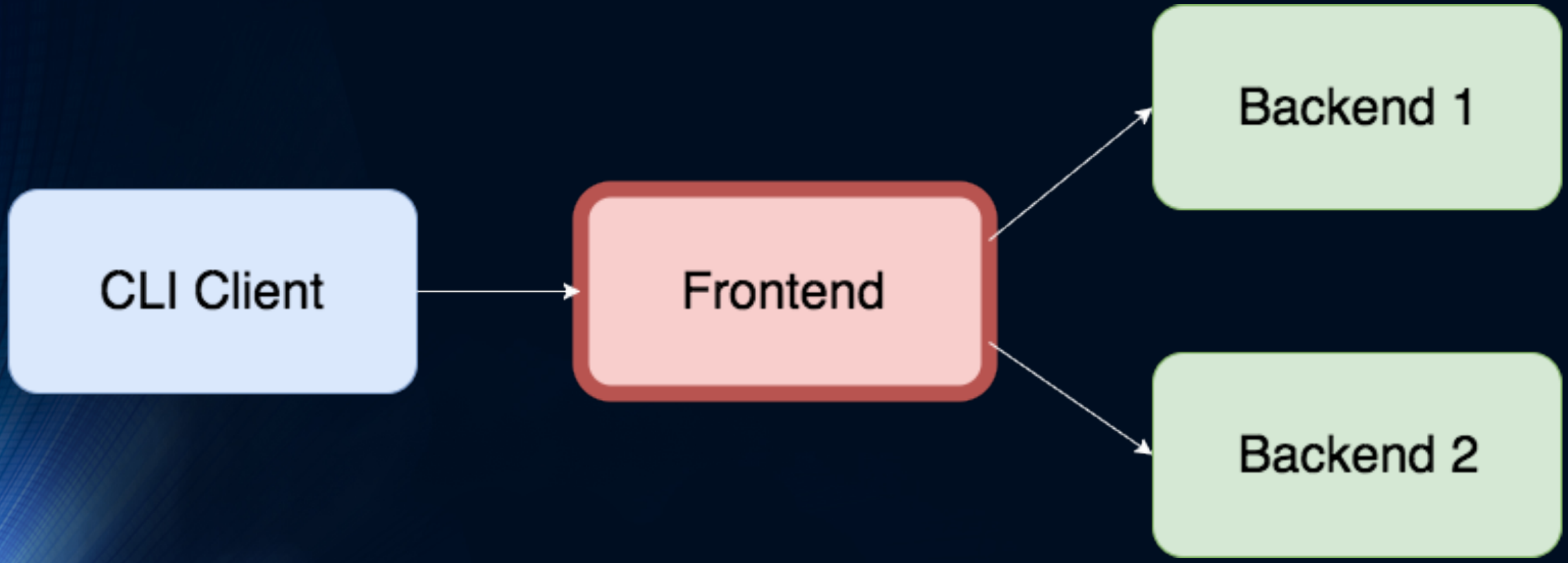
Source code – backend – streaming



Source code – Go backend - streaming

```
// Watch returns a stream of results identifying the query and this
// backend, sleeping a random interval between each send.
func (s *server) Watch(req *pb.Request, stream pb.SearchEngine_WatchServer) error {
  ctx := stream.Context()
  for i := 0; ; i++ {
    d := randomDuration(1 * time.Second)
    logSleep(ctx, d)
    select {
    case <-time.After(d):
      err := stream.Send(&pb.Result{
        Title: fmt.Sprintf("result %d for [%s] from backend %d", i, req.Query, *index),
      })
      if err != nil {
        return err
      }
    case <-ctx.Done():
      return ctx.Err()
    }
  }
}
```

Source code – frontend – streaming



Source code – Go frontend- streaming

```
// Watch runs Watch RPCs in parallel on the backends and returns a
// merged stream of results.
func (s *server) Watch(req *pb.Request, stream pb.SearchEngine_WatchServer) error {
  ctx := stream.Context()
  c := make(chan result)
  var wg sync.WaitGroup
  for _, b := range s.backends {
    wg.Add(1)
    go func(backend pb.SearchEngineClient) {
      defer wg.Done()
      watchBackend(ctx, backend, req, c)
    }(b)
  }
  go func() {
    wg.Wait()
    close(c)
  }()
  for res := range c {
    if res.err != nil {
      return res.err
    }
    if err := stream.Send(res.res); err != nil {
      return err
    }
  }
  return nil
}
```

Source code – Go frontend- streaming

```
// watchBackend runs Watch on a single backend and sends results on c.
// watchBackend returns when ctx.Done is closed or stream.Recv fails.
func watchBackend(ctx context.Context, backend pb.SearchEngineClient, req *pb.Request, c chan<- result) {
  stream, err := backend.Watch(ctx, req)
  if err != nil {
    select {
    case c <- result{err: err}:
    case <-ctx.Done():
    }
    return
  }
  for {
    res, err := stream.Recv()
    select {
    case c <- result{res, err}:
      if err != nil {
        return
      }
    case <-ctx.Done():
      return
    }
  }
}
```

gRPC – ecosystem

gRPC - ecosystem

- **grpc-gateway** - gRPC to JSON proxy generator following the gRPC HTTP spec
- **grpc-prometheus** - Prometheus monitoring for your gRPC servers.
- **go-grpc-middleware** - Golang gRPC Middlewares: interceptor chaining, auth, logging, retries and more.
- **grpc-opentracing** - OpenTracing is a set of consistent, expressive, vendor-neutral APIs for distributed tracing and context propagation
- **Polyglot** - A universal grpc command line client


GRPC

Summary

Use gRPC when:

- Efficiently connecting polyglot services in microservices style architecture – implementations in (**Java**, **Go**, **C**, C++, Node.js, Python, Ruby, Objective-C, PHP, i C#)
- Connecting mobile devices, browser clients to backend services
- Generating efficient client libraries
- Creating low latency, highly scalable, distributed systems.

gRPC is the answer to most of the microservices problems

Thank you!