GoTalks 27.3.

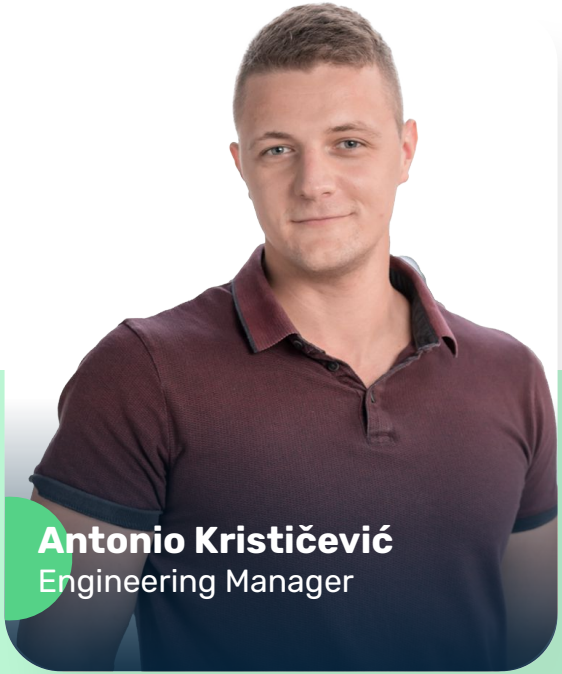
Powered by Fonoa

fonoa

Agenda

- Intro
- Go vs .NET
- Async service comms
- Q&A with 🍔🍺

Who are we?

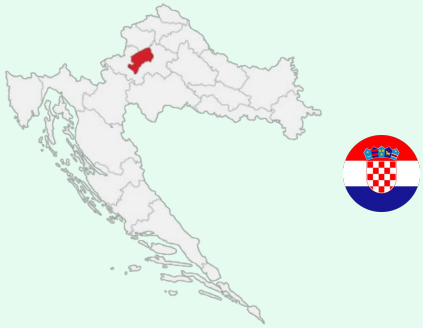


Antonio Krističević
Engineering Manager

Dražen Mrvoš
Senior Software Engineer

Fonoa

Osnovani 2019 u Zagrebu

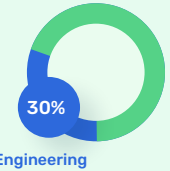


~$85 million

Investicije od VC-a

130+

Zaposlenika



100+ deployments

po danu

Stack

Backend

Go, .NET, node, Java

Frontend

NEXT.JS

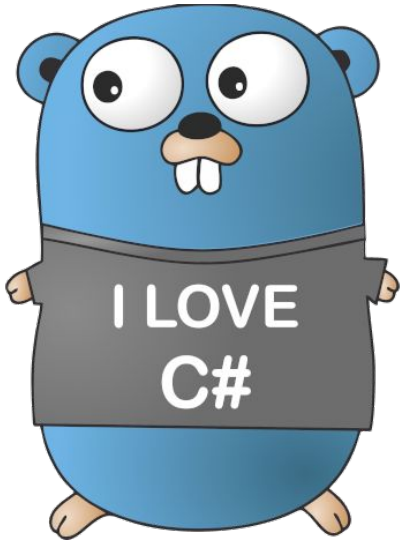
DB Engine

PostgreSQL

Infrastructure

in-house cloud-agnostic k8s

Golang from the perspective of .Net developer



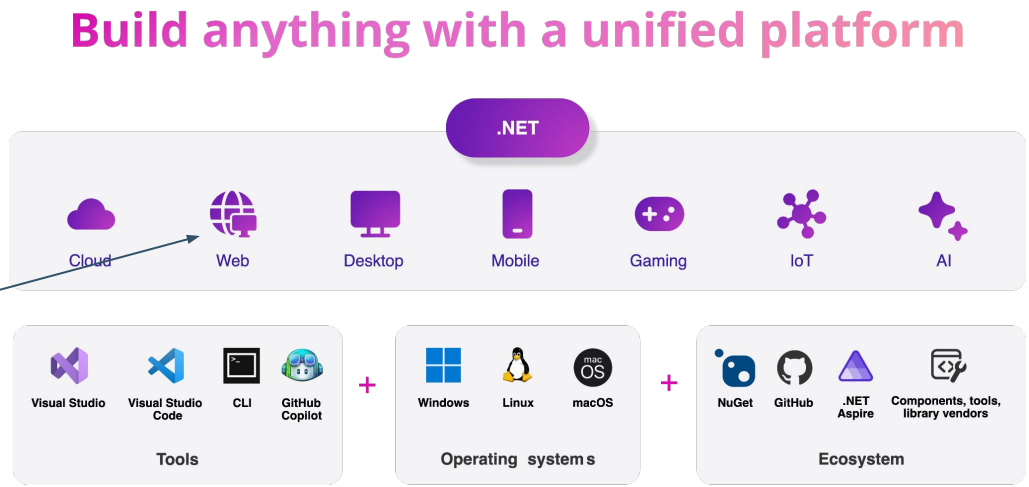
Ecosystem (.Net)

.NET is massive. Whether you need UI, APIs, web apps, or machine learning — it's all built into the framework, often with multiple overlapping ways to do the same thing.

That's both a pro (mature, stable, lots of options) and a con — Microsoft sets the direction, and that direction has shifted over time.

Take building web apps as an example:
- ASP.NET Web Forms
- ASP.NET (Core) MVC
- Razor Pages
- Blazor (SignalR vs WASM)

Fun fact - most of these are still supported! While flexible, this creates a landscape where developers have to constantly re-learn or migrate to the latest recommended approach.

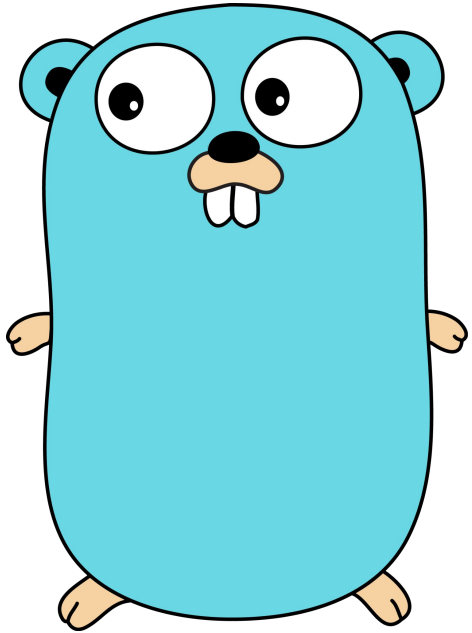


Ecosystem (Go)

Go doesn’t try to be a jack of all trades (and master of none).

Need a web server? You have it.
Want something fancier (fancier like having pattern-matching)? Build it yourself or grab a third-party lib.

You don’t get five competing frameworks — you get a solid, battle tested starting point and full control.



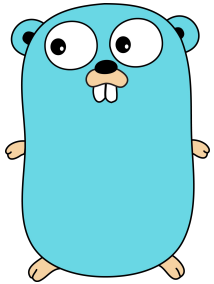
Tooling

Tooling has come a looong way since the ol' days of .Net framework:

- .NET Framework: Windows + Visual Studio only
- .NET Core: Cross-platform, CLI tooling starts appearing
- .NET 5+: CLI tools are first-class: `dotnet build`, `dotnet restore`, `dotnet ef`, etc.

Still, there's no built-in formatter or linter. You need to rely on third-party tools such as `Resharper`.

.NET



- Step 1: install Go
- Step 2: done

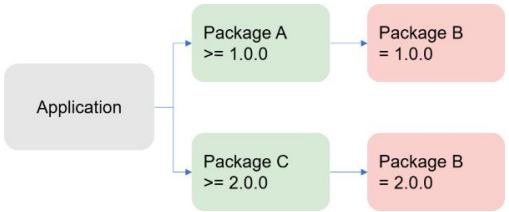
After these two steps:
- go build, go test, go run, go mod, etc. — all standard and ready to go
- Formatting and linting included (gofmt, go vet)
- No solution/project files, no configuration — your folder is the project

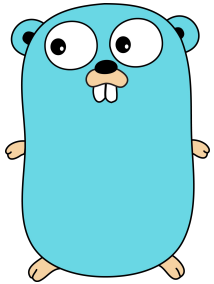
Dependency resolution

- In past we had DLL hell 😈. We used to resolve dependencies (when needed) through binding redirects that could cause actual hell if you weren't careful in what you're doing

```
<dependentAssembly>
  <assemblyIdentity name="Newtonsoft.Json"
publicKeyToken="30AD4FE6B2A6AEED" culture="neutral"/>
  <bindingRedirect oldVersion="0.0.0.0-13.0.0.0" newVersion="13.0.0.0"/>
</dependentAssembly>
```

- Today, things have gotten a lot better, but still dependency resolution issues may pop-up - these days, they should cause build errors instead of runtime errors.
- There's no built-in equivalent of go mod tidy





"A little copying is better than a little dependency"
– Go proverbs

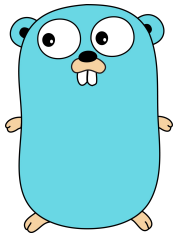
- `go.mod` + `go.sum` ensure consistent versions across different environments
- `go mod tidy` takes care of unused dependencies

Dependency injection

- Dependency Injection is basically a standard of how dependencies are resolved in modern .NET
- Services are registered in a container with specific lifetimes (transient, scoped, singleton), and then injected automatically where needed

```
services.AddSingleton<IAppConfiguration,
AppConfiguration>();
services.AddScoped<ISomeService, SomeService>();
```

.NET



- Dependencies are passed explicitly
- No lifetimes, no magic, no debugging weird resolution issues
- No runtime errors stating dependency cannot be resolved

```
type NotificationService struct {
    email EmailService
}

func NewNotificationService(email EmailService) Service {
    return Service{Email: email}
}
```

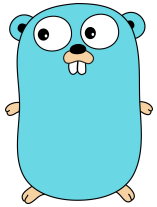
Testing

- Testing relies on external frameworks such as `xUnit`, `NUnit` or `MSTest`. Same goes for benchmarking (`BenchmarkDotNet`)
- Usually, test methods have specific attributes so that they may be discovered as such
- Rich ecosystem (`Moq`, `FluentAssertions`, …)
- Since tests are separate projects - publicly accessible methods/properties may be tested (by default)

```
[Theory]
[InlineData(2, 2, 4)]
[InlineData(3, 5, 8)]
[InlineData(10, -2, 8)]
public void Add_ShouldReturnExpectedSum(int a, int
b, int expected)
{
    var result = Add(a, b);
    result.Should().Be(expected);
}
```

.NET

- tests are just Go code like any other: testing is a first-class citizen. No imports, no test runner - write `*_test.go` files and you're good to go
- Tests may live within the same module so internals may be tested as well



```
func TestAdd(t *testing.T) {
    tests := []struct {
        a, b     int
        expected int
    }{
        { 2, 2, 4 },
        { 3, 5, 8 },
    }

    for _, tt := range tests {
        got := Add(tt.a, tt.b)
        require.Equal(t, tt.expected, got)
    }
}
```

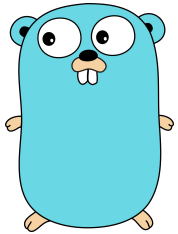
fonoa

Error Handling Philosophy

- Structured error handling using try/catch
- Many ways to `throw` and many ways to `catch`
- Easy to conditionally handle exception using `when` operator

```
try {
 // do something that throws exception
}
// now what?
catch (Exception ex) { … }
catch (Exception ex) when (e is not SomeException) { … }
catch (Exception ex) { …; throw; }
catch (Exception ex) { …; throw ex; }
catch (Exception ex) { …; throw new OtherException() }
finally {
  // executed always
}
```

.NET

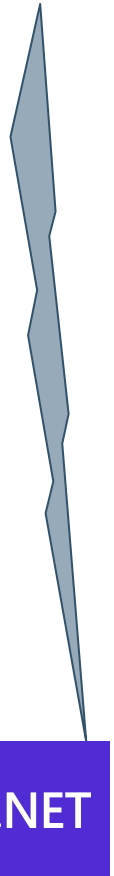


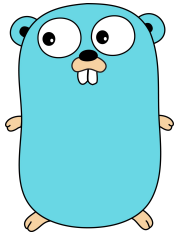
In Go, error handling is boring — and boring is good.

Concurrency/parallelism

- Provides a powerful toolbox to handle both parallelism and asynchronous workflows:
  - `Parallel.For` / `Parallel.ForEach` — for CPU-bound parallel processing
  - `Task`, `Task.WhenAll`, `Task.WhenAny` — to coordinate multiple units of work
  - `async/await` — for non-blocking I/O and simplified async programming
  - Thread-safe collections and primitives — like `ConcurrentDictionary`

- Provides a lot of misconceptions about them:
  - If I make my code async, it'll run faster
  - Starting CPU-bound work with `Task.Run()` and immediately awaiting it — adds thread-pool overhead without real benefit
  - Confusion between parallelism (multiple things at once) and asynchrony (not blocking)





Concurrency-first approach:

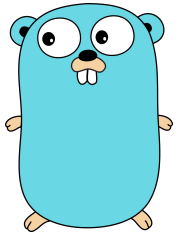
- Goroutines — lightweight thread-like constructs managed by the Go runtime
- Channels — "Do not communicate by sharing memory - share memory by communicating."
- No thread pool, Go runtime handles scheduling
- You think in terms of concurrent processes, not threads

Syntactic sugar

- C# is rich in sugar and you get addicted to it: null conditional operator, LINQ (method or query syntax), pattern matching, null coalescing operator, …

```
string displayName = user.Name ?? "Anonymous";

var activeUsers = users.Where(u => u.IsActive);

var activeUsers =
    from u in users
    where u.IsActive
    select u;

if (obj is string s && s.Length > 0)
{
    Console.WriteLine($"Non-empty string: {s}");
}
```

.NET

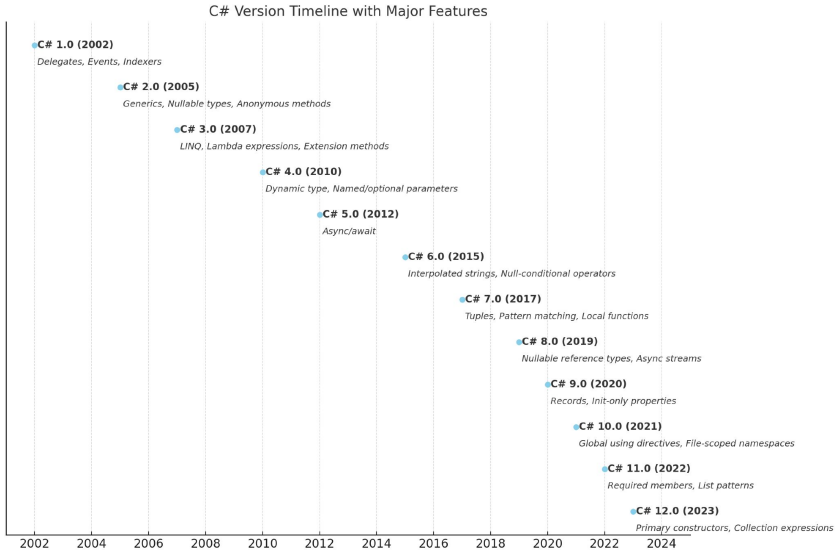


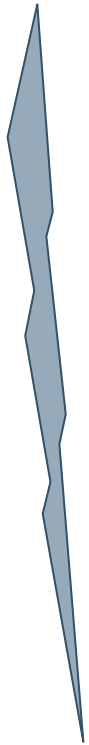
Go is verbose, which is good or bad - depending on who you ask :)

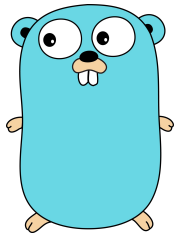
```
displayName := "Anonymous"
if user.Name != "" {
    displayName = user.Name
}

activeUsers := make([]User, 0, len(users))
for _, u := range users {
    if u.IsActive {
        activeUsers = append(activeUsers, u)
    }
}

if s, ok := obj.(string); ok && len(s) > 0 {
    fmt.Printf("Non-empty string: %s", s)
}
```

Language Evolution

- C# and .Net evolve fast
- While versatile and powerful — it can feel like a moving target
- Also, sometimes, it feels like learning is actually learning the C#/.Net specific quirks instead (knowledge useless outside that domain):
  - what does modifier `protected internal virtual` mean.
  - what `ConfigureAwait(false)` does
  - `async void` is bad. Why?
  - `override` vs `new` and the consequences



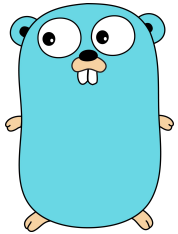




- Go evolves *slowly and deliberately*
- It took 10ish years to get generics :)

Decimal support

- C# - `decimal` type is the first class citizen
- Yeah, just filling the content but not much else to say - just use it



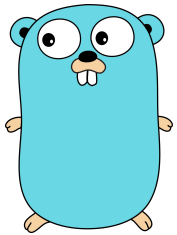
- Different teams tackled the same issue differently
  - Some use `shopspring/decimal`
  - Some use their own decimal type (currently at least three different in use)
  - Some use `int64` and make sure that 10^x factor is applied

ORM

ORM options are mature and varied:

- Entity Framework for full-blown mapping
- Dapper for micro ORM needs

```
// Dapper
var user = await
conn.QueryFirstOrDefaultAsync<User>("SELECT *
FROM users WHERE user_id = @Id", new { Id = 1 });

// EF
var user = await context.Users.Where(u => u.UserId ==
1).FirstOrDefaultAsync();
```



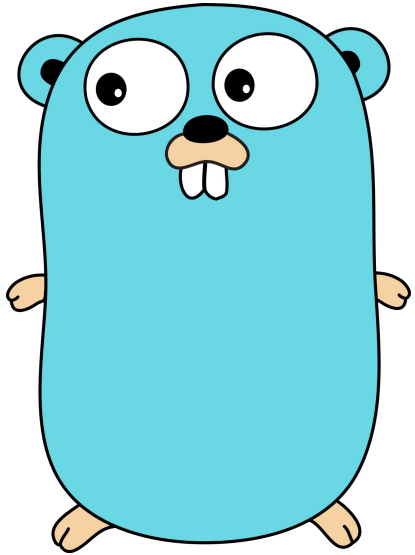
Exist, but are not widely adopted by our teams (this is something in to-do). Sooo, for now - we occasionally end up in situations such as:

```
where ($2::text is null OR LOWER(cm.file_name) LIKE '%' ||
LOWER($2) || '%')
and ($3::text is null OR t.tax_country = $3)
... x10
and ($14::uuid is null OR t.alternate_id = $14)
```

Go@Fonoa

We love Go, not because it is the best thing in the world but because:

- It gets the job done
- Good enough for all our use-cases - services, CLI tools
- Developers of different profiles get used to it quickly
- Based on our experience, it is easier to hire new Go devs than .Net devs
- Performance-wise: we're aware that we could probably achieve similar results with other languages as well



fonoa

Async service communication

Agenda

- History lesson
- Problems to solve
- Core idea
- Design and implementation
- Challenges
- Rollout and adoption
- Positive side effects
- FAQs

agenda

- History lesson
  - Shape of our platform, products and services
  - Evolution and fast expedrimentation in silos
  - Vision is always there - single platform
- What problems exist when building in silos
- What is the nature of our industry?
  - Compliance for many years to come
- Solve problems only once at platform level
- Enter message bus
- Core idea behind message bus
- Implementation
  - Design (constraints that we set, components)
  - Challenges
  - Tradeoffs
  - Future and extensibilkity
- Rollout to all teams (libarries etc_ar
- Driving adoption within the organiza
  - Why would we use it?
- Side-effects of adoption
  - Ledger, megafonoa, emails, etc
  - You control the roads and you
- FAQs

History

Why are we even doing this?

History

Vision - build an indirect tax platform

Execution - many products in different phases of growth

Looking for **PMF** in isolation.

Speed of Execution > Product cohesion

Did we do the right thing?

CHALLENGE

fonoa

01 Solving the same problem many times

02 Meeting the SLA

03 Data durability

04 Platformization

Philosophy

Hammock-driven development

01

Simple and lean solution - solve a real problem

02

Own the roads

03

Have a time machine

04

What do we actually need?

- **Segregation of ingestion and processing**
- **We set up tiered SLOs**
  - T1 - 99.5% transactions are ingested successfully within 1s
  - T2 - 99.99% transactions are processed successfully (either success or failure)
- **Teams are already using their flavour, how do we standardize this?**
- **What do we really need it to do?**

fonoa

Requirements

Cloud agnostic

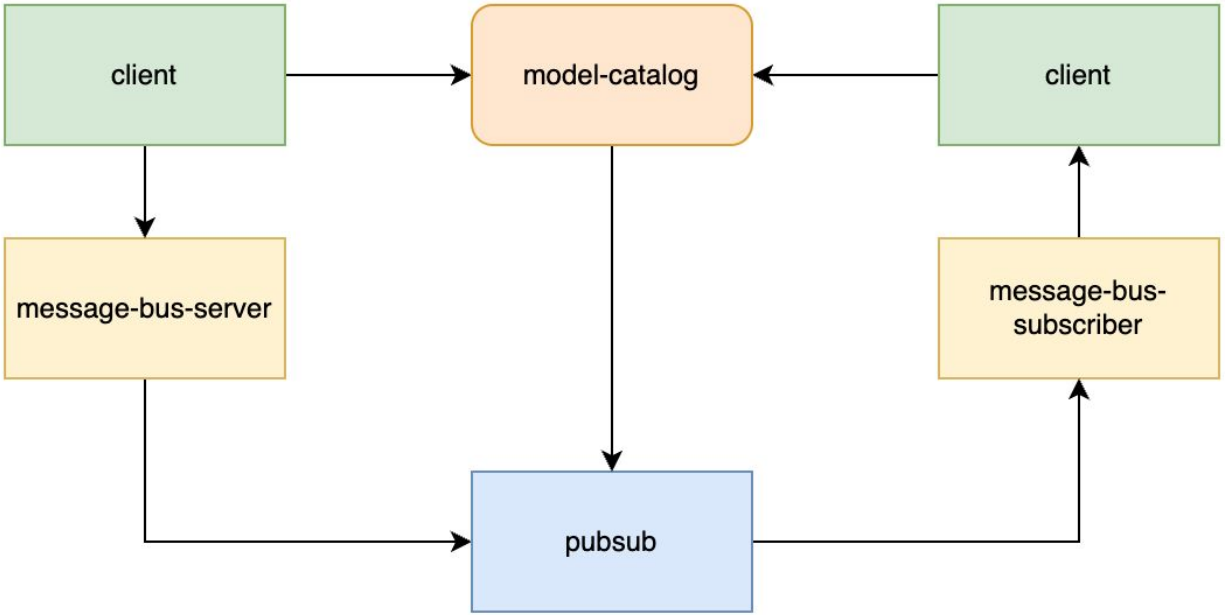
Performance and reliability

Cataloging ability

Platform-wide

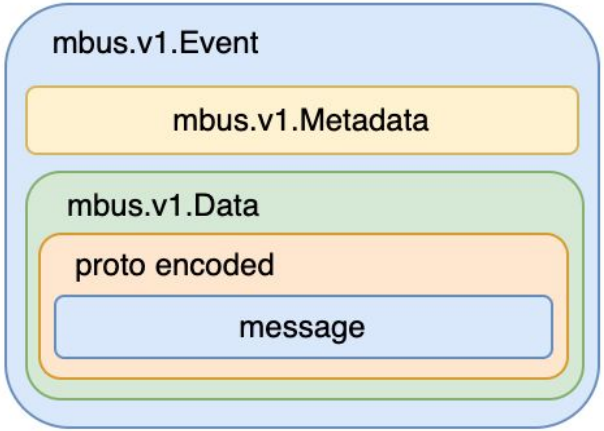
fonoa

High-level flow



Model catalog

- Schema definition for all messages passed to the message bus
- Enforce required metadata through the schema
- **Protobuf** format
  - binary encoding
  - Enforce backwards compatibility
- Autogenerated client libraries for GO, TS, C#



Server

- Thin abstraction for the transport layer
- Decouples publishers from underlying technology
- Narrower interface and guarantees
- Metadata enforcement
- Does not leak underlying technology choices
- Simple gRPC API
- Automatically infers the topic based on message type

```
syntax = "proto3";

package mbus.v1;

service MessagePublisherService {
  rpc PublishMessage(PublishMessageRequest) returns (PublishMessageResponse)
}

message PublishMessageRequest {
  Event event = 1;
}

message PublishMessageResponse {
  bool ack = 1;
  string message_id = 2;
}

message Event {
  Metadata metadata = 1;
  // data is the payload model encoded to protobuf
  bytes data = 2;
}
```

Subscriber

- Inverse flow -
  - Clients initiates
  - Server sends messages
  - Client ACKS
- Bi-directional gRPC stream
- Errors must be handled within the message
- gRPC stream is terminated in case of error
- Topic, subscription, and client are automatically inferred

```
service MessageSubscriberService {
  // Streaming required so the server doesn't need to know where to send requests to
  rpc SubscribeMessage(stream SubscribeMessageRequest) returns (stream SubscribeMessageResponse)
}

message SubscribeMessageRequest {
  SubscribeRequestType request_type = 1;
  // Fully qualified type name, e.g. models.test.v1.TestMessage
  string type_name = 1;

  // Acknowledge parameters
  string ack_id = 2;
  // This should be true to acknowledge the message, false to "nack" and
  // immediately refuse acknowledging. "nack" should only be sent in case of a
  // transient error as the message will be resent as is.
  bool acknowledge = 3;
}

enum SubscribeRequestType {
  SUBSCRIBE_REQUEST_TYPE_UNSPECIFIED = 0;
  SUBSCRIBE_REQUEST_TYPE_INITIATE = 1;
  SUBSCRIBE_REQUEST_TYPE_ACK = 2;
  SUBSCRIBE_REQUEST_TYPE_TERMINATE = 3;
}

message SubscribeMessageResponse {
  SubscribeResponseType response_type = 1;

  // Message parameters
  Event event = 2;
  // The ID to send to confirm message acknowledgement
  string ack_id =3;

  // Error parameters
  string error = 4;
}

enum SubscribeResponseType {
  SUBSCRIBE_RESPONSE_TYPE_UNSPECIFIED = 0;
  SUBSCRIBE_RESPONSE_TYPE_MESSAGE = 1;
  SUBSCRIBE_RESPONSE_TYPE_ERROR = 2;
}
```

Subscriber challenges

1

Discovery problem

How does subscriber know where to send messages? Alternatives considered?

2

Sticky ACKs

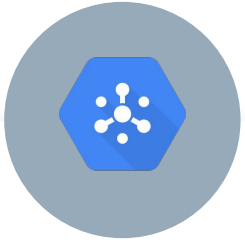
Clients must send ACK to same instance (in-memory ack map)

3

Load balancing

Messages are not balanced, stream connections are.

Transport



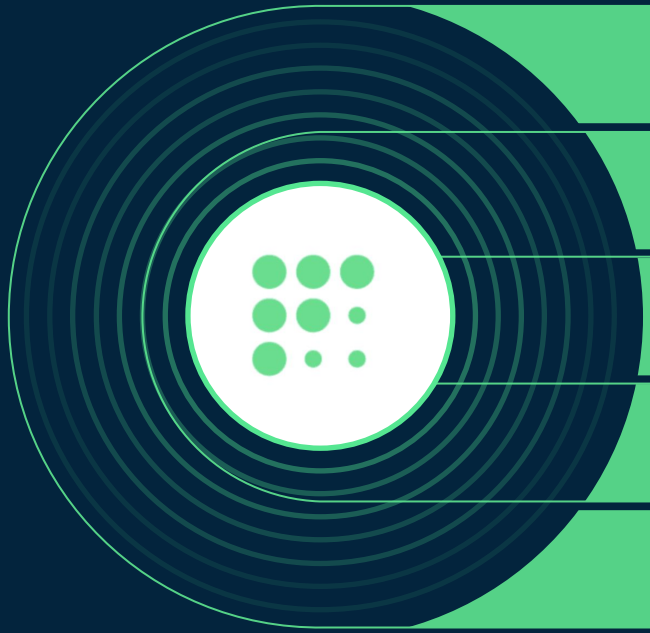
Pubsub

- Emulator is terrible
- Unavailable in some regions
- Wide feature set

NATS.io

- Used in acceptance tests
- User in regions where GCP is unavailable
- Potential future

Technical considerations

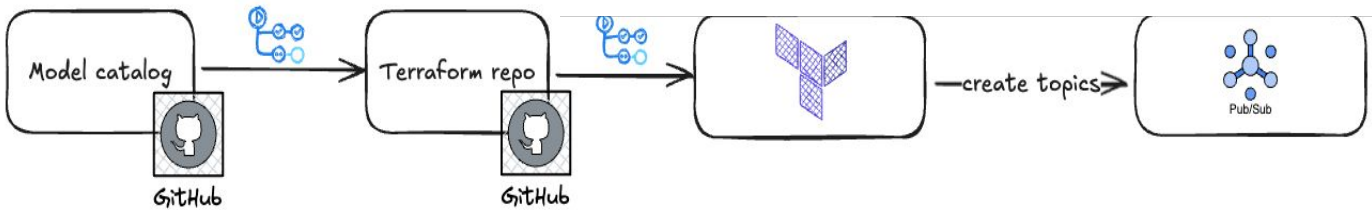


At least once delivery	Exactly once is harder to pull off
N:M delivery	One topic per model, multiple producers and consumers
Scalability & Resilience	Supports company growth
Accountability	Topic subscribers should be recorded so publishers know who they are
Immutability	Truthful record of historical data

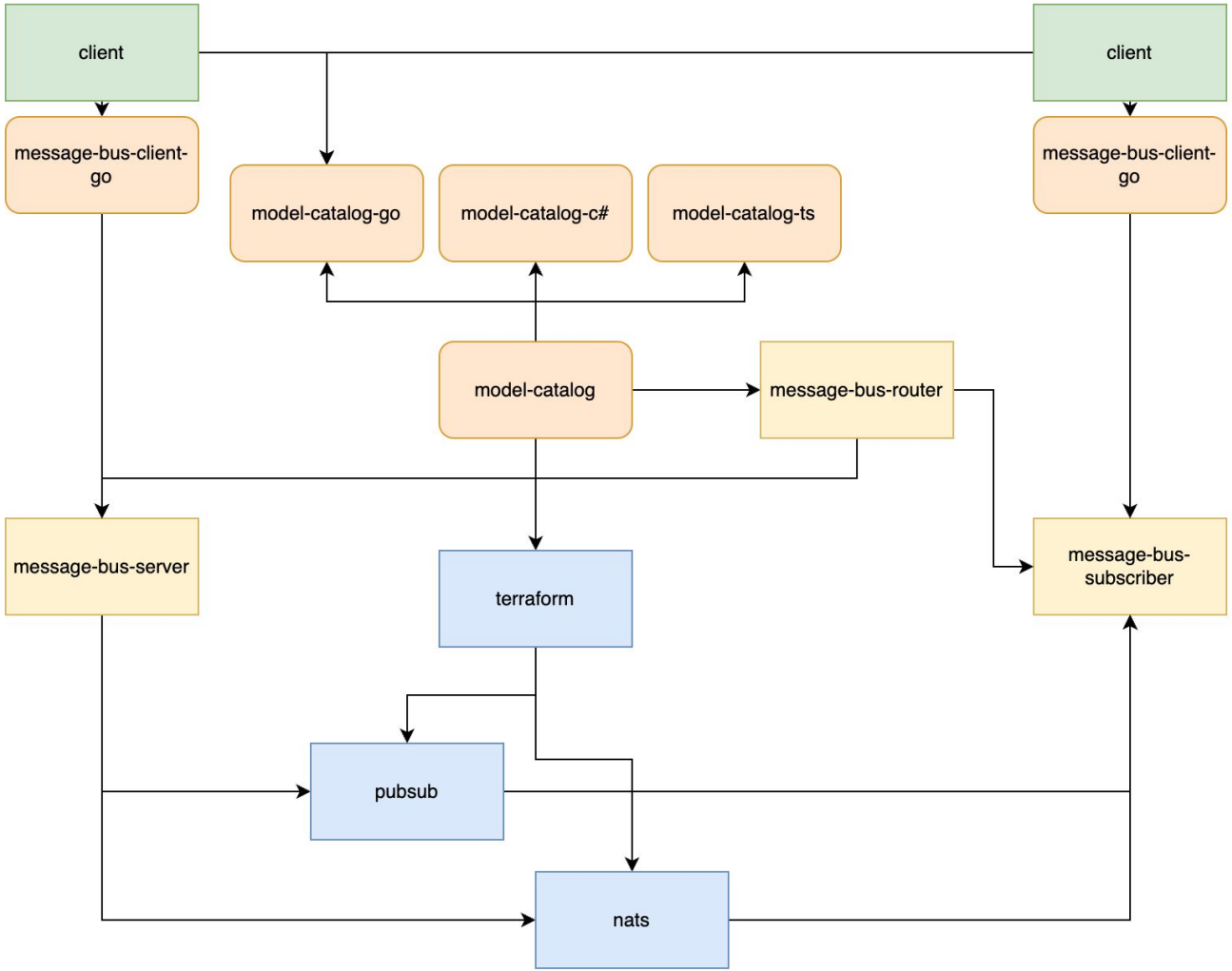
Topic creation

- Topics associated with message definition across all environments
- Subscribers defined per environment
- GH Actions automatically create everything

```yaml
environments:
  - dev
  - production
teams:
  - alpha
  - beta
topics:
  models.test.v1.TestCountryMessage:
    owner: alpha
    subscribers:
      foo-api:
        environments:
          - dev
        countries: ["BE", "JE"]
```



Full picture



Rollout & Adoption

- Multi-cloud challenges
- Different internal technologies
- Client requests accelerating adoption
- Chaos testing

fonoa

Usage - Publish a message

```
publisher, err := publisher.NewClientBuilder(serviceName, mbURI).
    WithMetrics(statsdClient).
    WithBackoff(3, 100). // 3 retries with 100ms initial backoff
    Build();
defer publisher.Close()

if err != nil {
    log.Err(err).Msg("cannot create messagebus client")
}

publisher.Publish(ctx, myProtoMessage, myMetadata)
```

Usage - Subscribe to a topic

fonoa

```
subClient, err := subscriber.NewClientBuilder(serivceName, mbURI).
    WithMetrics(metrics).
    Build()
defer subClient.Close()
if err != nil {
    log.Fatal().Err(err).Msg("Cannot start MB subscriber client")
}

sub, err := subscriber.NewSubscriptionBuilder(mbClient, myHandler).
    WithServiceName(serviceName).
    WithConcurrentProcesses(10).
    Build()
defer sub.Stop(ctx)
if err != nil {
    log.Fatal().Err(err).Msg("Cannot start MB subscriber for TestTopic")
}
```

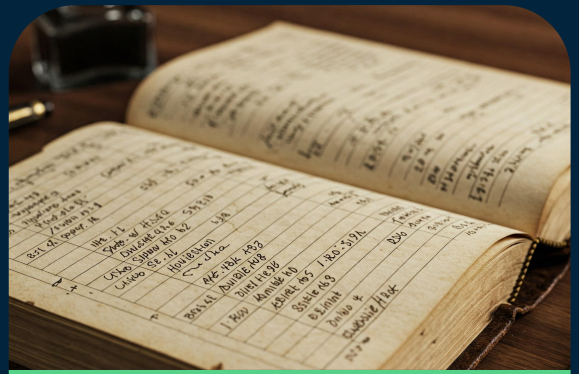
Usage - Handler signature

```go
type Process[M proto.Message] func(ctx context.Context, msg M, metadata *mbusv1.Metadata) error
```

```go
func (h *myHandler) HandlerFunction(ctx context.Context, myModel *foo.BarModel, metadata *mbusv1.Metadata) error {
    // Do something

    return nil // nil automatically ACKs the message, err of any kind NACKs it
}
```

fonoa

Positive side-effects



Ledger
Long-term immutable storage

Notification system
webhooks, emails, whatever

Time machine
Build new products with existing data

FAQs

fonoa

Why not Kafka/SNS/NATS?

Because we already used pubsub and had supporton infra level.

Why at least once delivery?

Generally speaking more complex and comes with asterisks, idempotency was simpler for us.

Does it have message leasing?

No, there were needs here and there but not enough to justify.

Can you tune retry logic?

Yes but no. If you need specific retry logic consider building it into a product.

Cross-cluster access?

Yes but please don't. We had to do this while moving away from Azure.

Does it have a DLQ?

It used to. No one used it. We deprecated it.

Thank you!