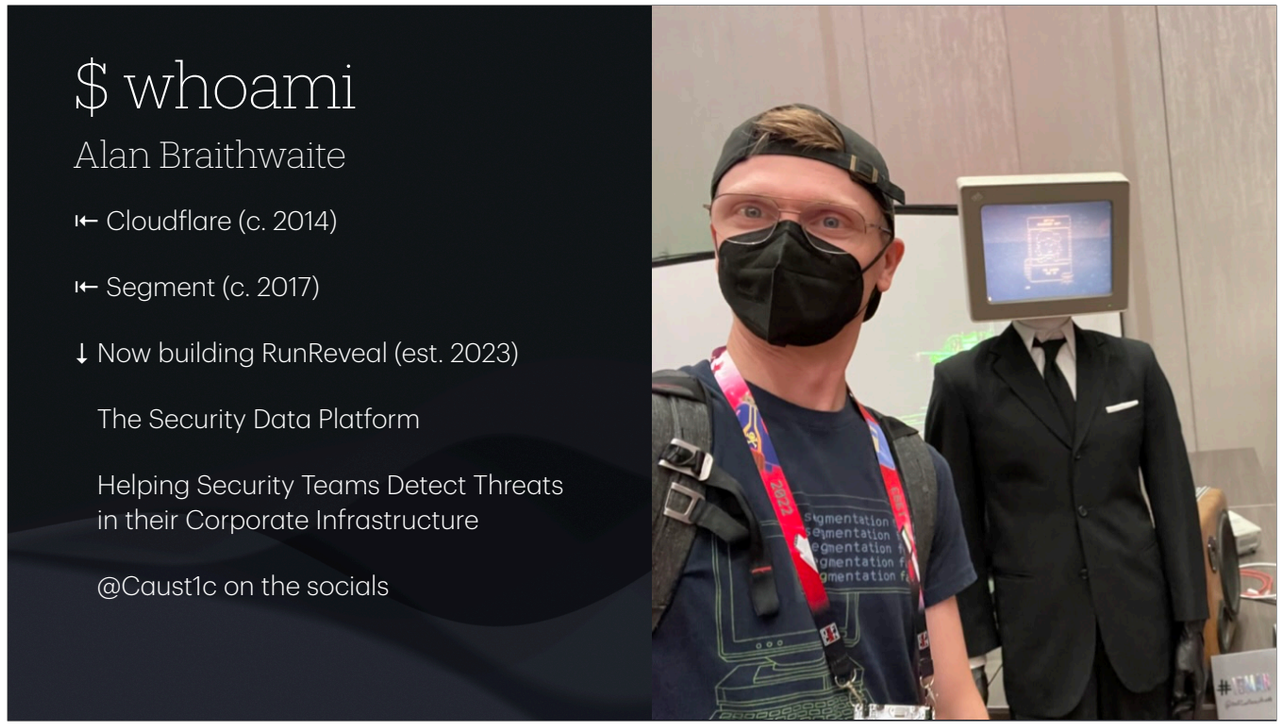
Processing Millions of Events per Second Reliably using Generics

Or, how I learned to love logs by embracing Go

Alan Braithwaite — GopherCon 2024



Aside from the 10 year anniversary of GopherCon, it’s also my 10th year of programming Go professionally, so this GopherCon is of particular significance to me.

Around that time, I was absorbing as much information on scalable systems operations.
Observability rabbit hole which led to learning about centralized logging software
Learned about logstash, flume, fluentd, graylog, syslog, splunk.
What I didn’t know at that time was that I was about to land a position at Cloudflare building and operating one of the largest logging pipelines in the world. Keys to the porche and good luck! Segment / RunReveal

Imagine you're an ant in a snow globe the size of Jupiter. You're tasked with catching every snowflake and sorting them by size and shape before they hit the ground.

That's the experience of processing data at scale.

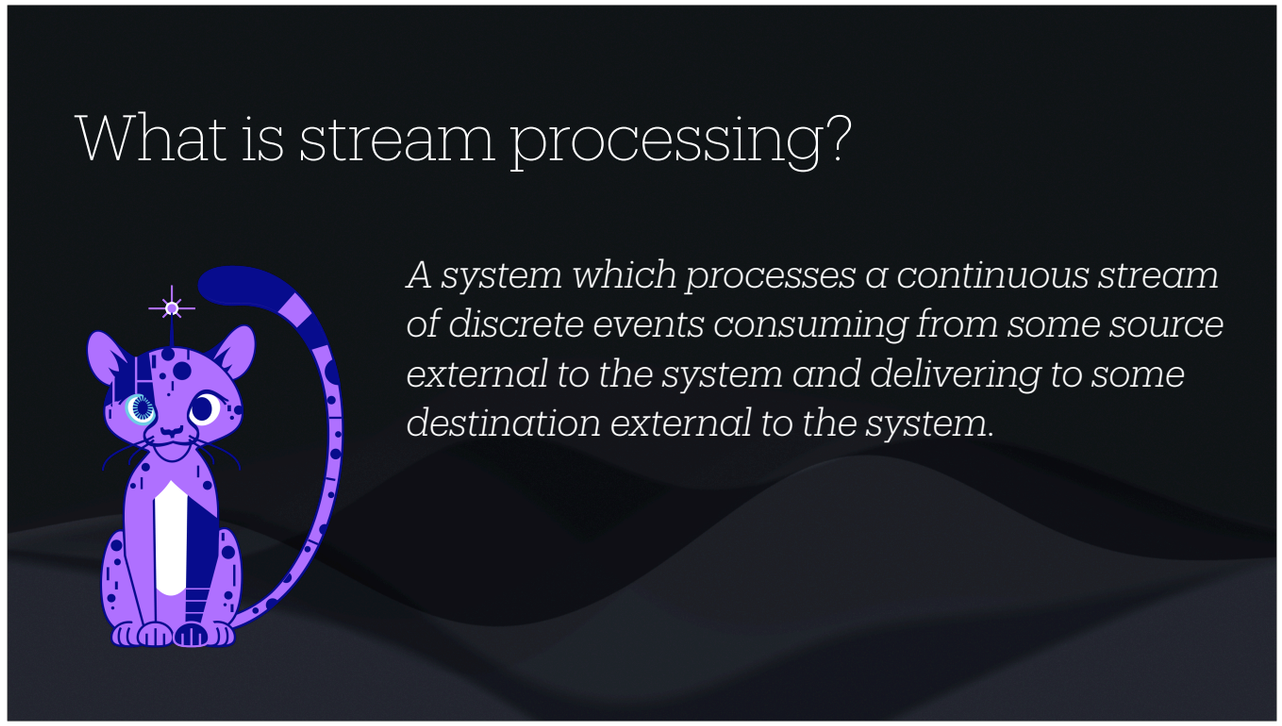
— Albert Strasheim

This is what my manager had to say about what we did at Cloudflare on the data team.

The culmination of my learnings from working at CloudFlare and at Segment is a stream processing library called Kawa.

In this talk we'll cover the why I wrote kawa, how it's used, and as the design and implementation that bring it all together.

But first,



External sources and destinations can include APIs, Webhooks, Messaging queues or even local storage like a file.

Also, I'll probably use "events", "logs" and "messages" interchangeably throughout the talk to mean the same thing. Apologies for that. It comes from reading so many different implementations for these things.

Why another framework?
Most weren't written in Go. — all the logging systems previously mentioned and also looked at Flink, Spark, Druid, etc.
Benthos was a project that probably came closest to Kawa's goals, but I didn't like the plugin architecture at the time I learned about it, but I also really wanted to use generics for reasons that will become clear.

Also Benthos is now RedPanda Connect, and has also been forked… Typical OSS drama ensued.

Goals

- Fast
- Plugins should be relatively easy to create
- Support any serializable data types
- Batteries included features
- At least once deliveries

Non-Goals

- Slow
- Directed Acyclic Graph Execution
- Exactly Once Deliveries

Goals:
Fast (~10-100k events/s per processor for simple workflows)
Simple to write plugins
Support multiple/any data types
Batteries included runtime processing & batching destination primitives
At least one delivery guarantees

Exactly Once deliveries can be effectively achieved in practice with at least once with deduplication.

Those goals are largely informed by experience and need.

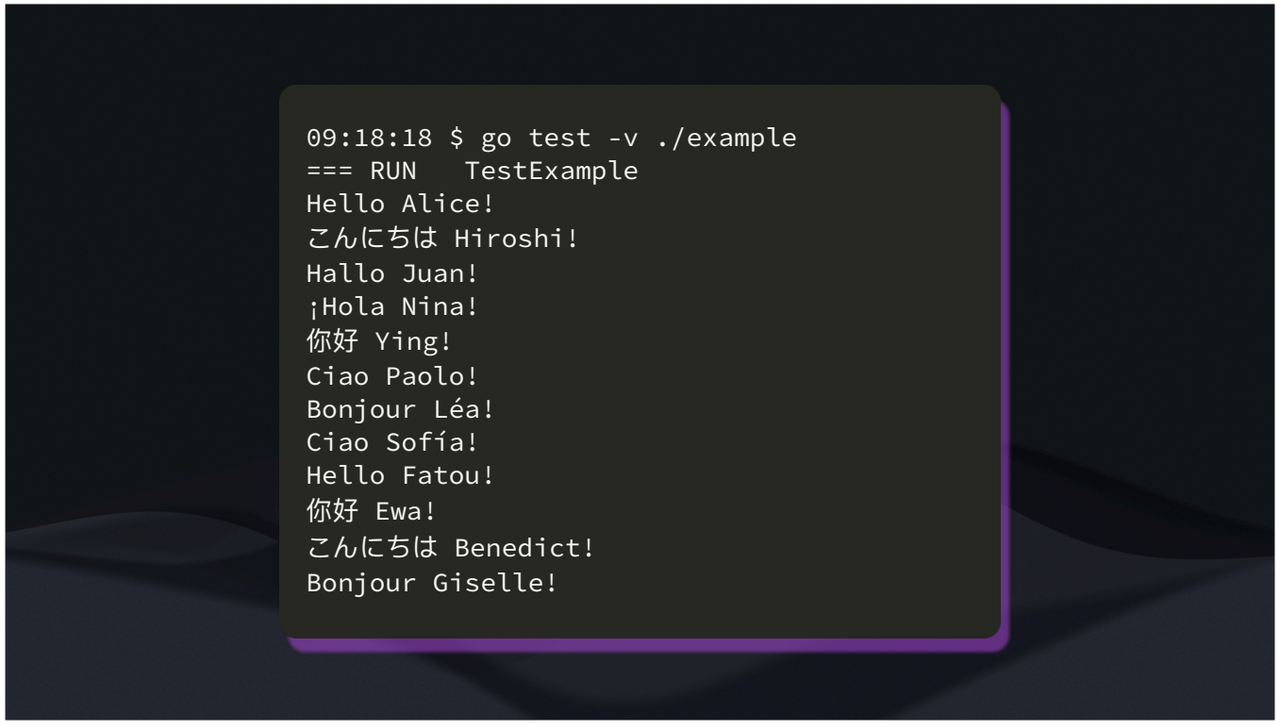
Experience tells me that there's no perfect event queues out there, they've all got major tradeoffs.
There are many, they have different trade offs and properties.
The point is whatever it is today, it's probably going to change as the company evolves, and you almost certainly don't want to start with Kafka.
And we need to support multiple types. Can't count how many times I've copied a bunch of code just to support another type, only for it to get missed in bug fixes later.

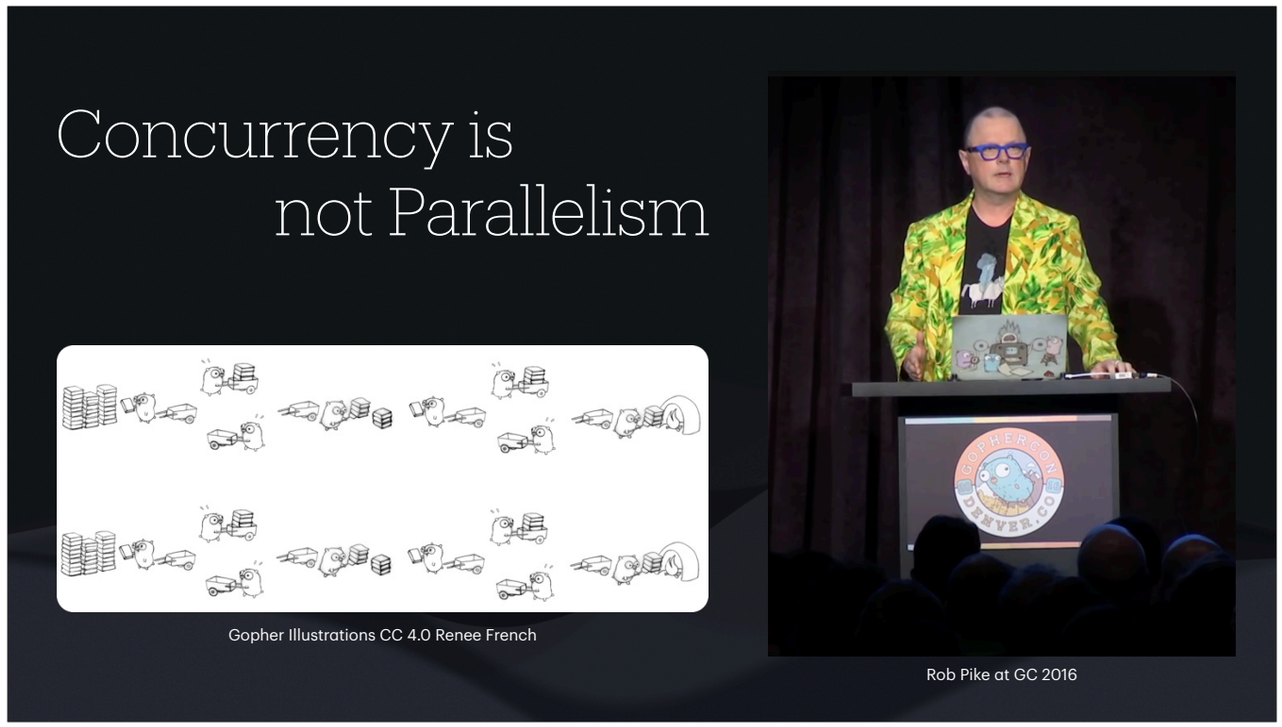
This is our product, all of these sources have different types that need to be deserialized and normalized. Generics help considerably.

```go
type Folk struct {
    Name string `json:"name"`
    Lang string `json:"lang"`
}

const folks = `{ "name": "Alice", "lang": "EN" }
{ "name": "Hiroshi", "lang": "JA" }
{ "name": "Juan", "lang": "DE" }
{ "name": "Nina", "lang": "ES" }
// …
{ "name": "Ying", "lang": "ZH" }
`

var greetings = map[string]string{
    "EN": "Hello %s!",
    "JA": "こんにちは %s!",
    "DE": "Hallo %s!”,
    // …
}
```

```
wg := await.New()
rdr := strings.NewReader(folks)
scanSource := scanner.NewScanner(rdr)
wg.Add(scanSource)
folkSource := JSONFolk{wrapped: scanSource}
printDest := printer.NewPrinter(os.Stdout)
k, _ := kawa.New[Folk, []byte](kawa.Config[Folk, []byte]{
    Source: folkSource,
    Handler: kawa.HandlerFunc[Folk, []byte](
        func(ctx context.Context, m kawa.Message[Folk]) ([]kawa.Message[[]byte], error) {
            greeting, ok := greetings[m.Value.Lang]
            if !ok {
                return []kawa.Message[[]byte]{}, nil
            }
            greeting = fmt.Sprintf(greeting, m.Value.Name)
            return []kawa.Message[[]byte]{{Value: []byte(greeting)}}, nil
        }),
    Destination: printDest,
})
wg.Add(k)
ctx := context.Background()
_ = wg.Run(ctx)
```

This might look a little involved, but when you've written these things dozens of times you'll begin to appreciate the things you don't have to write when building a processor like this.


09:18:18 $ go test -v ./example
=== RUN   TestExample
Hello Alice!
こんにちは Hiroshi!
Hallo Juan!
¡Hola Nina!
你好 Ying!
Ciao Paolo!
Bonjour Léa!
Ciao Sofía!
Hello Fatou!
你好 Ewa!
こんにちは Benedict!
Bonjour Giselle!

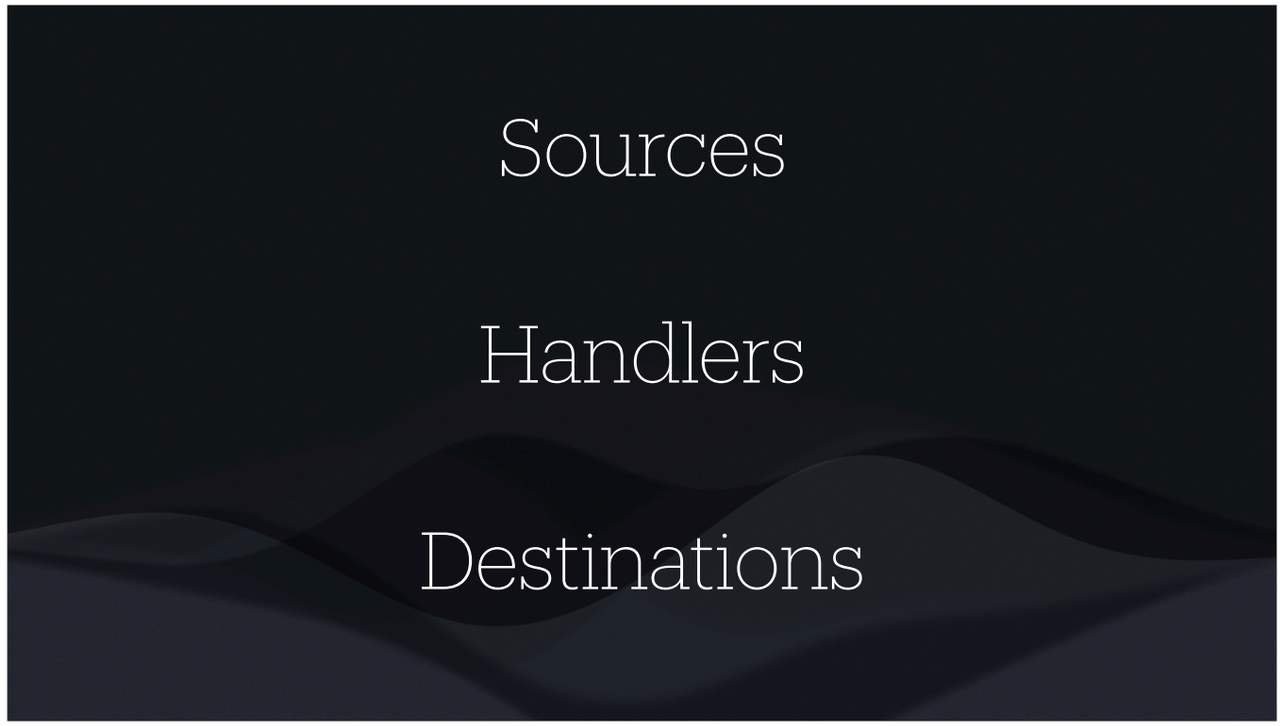


Gopher Illustrations CC 4.0 Renee French

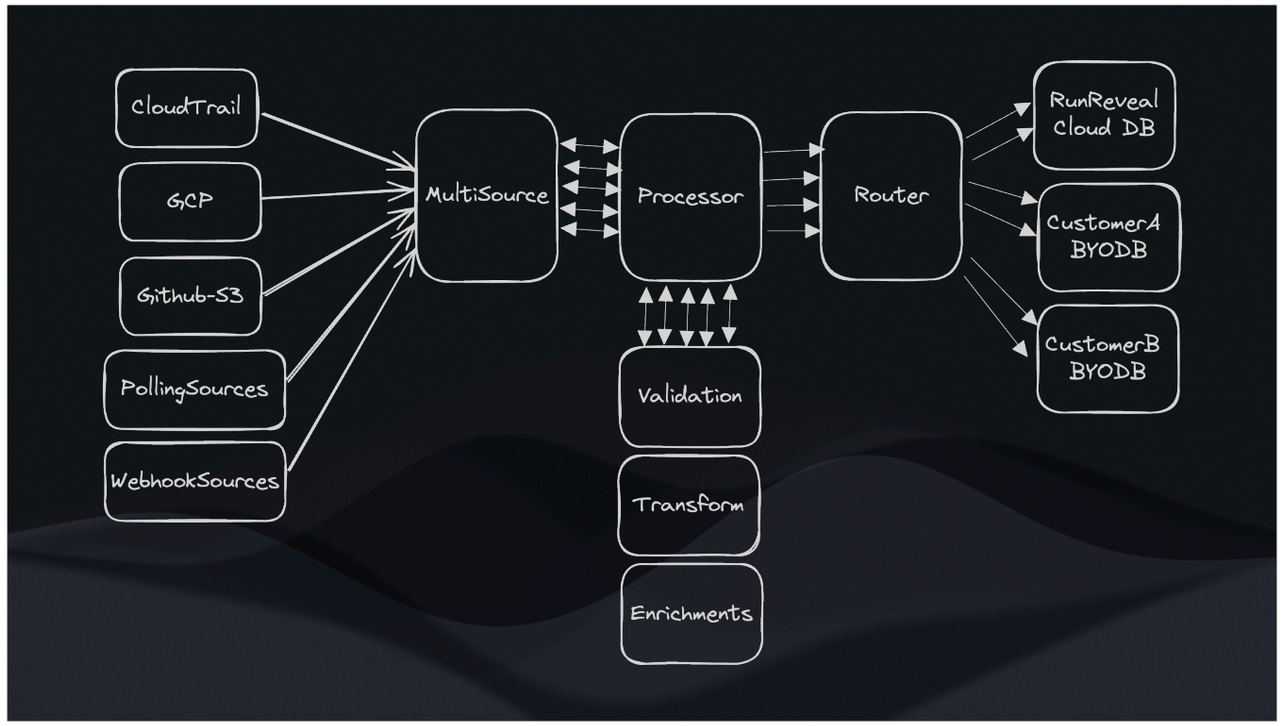
Rob Pike at GC 2016

Concurrency is not parallelism
Concurrency, not parallelism, is what leads to fast programs.
Parallelism applied without care can easily to high contention and slower programs.

Applying concurrent design patterns, we can break down the hard work, and parallelize individual components to optimize for the slowest parts of the pipeline.



The components, which all run concurrently enabling smooth pipelining of data.



Here’s a general overview of how we actually use kawa at RunReveal.

All the boxes on the left are sources, as is the multi source.

How many of you have written a channel multiplexer multiple times over because you had channels of different types?

The processor in the center manages the pipeline and after receiving an event from a source, sends it through the handlers before sending the processed event to it’s destination, the router which determines which external database to write the event to (also destinations).

The Message

```go
type Message[T any] struct {
    Key        string
    Value      T
    Topic      string
    Attributes Attributes
}

type Attributes interface {
    Unwrap() Attributes
}
```

For types that are data containers, I prefer concrete types over interfaces.
To me, an interface describes the behavior of a system, and a struct describes the attributes you expect some data to represent. Using a struct here as opposed to an interface makes programming and debugging the system much simpler.
With generics, we can now have concrete types that don’t have the same drawbacks of passing around interfaces.
Attributes work like context, allowing sources to communicate implementation details safely to other parts of the program that optionally care about those details without requiring that all systems know about them.

Sources

```
type Source[T any] interface {
    Recv(context.Context) (Message[T], func(), error)
}

type SourceFunc[T any] func(context.Context) (Message[T], func(), error)

func (sf SourceFunc[T]) Recv(ctx context.Context) (
  Message[T],
  func(),
  error
) {
    return sf(ctx)
}
```

SourceFunc inspired by http.HandlerFunc

Recv blocks until a message is returned, even if waiting forever.
This is a nice property because in that Recv function it’s

Destinations

```go
type Destination[T any] interface {
    Send(context.Context, func(), ...Message[T]) error
}

type DestinationFunc[T any] func(context.Context, func(), ...Message[T]) error

func (df DestinationFunc[T]) Send(
  ctx context.Context,
  ack func(),
  msgs …Message[T]
) error {
    return df(ctx, ack, msgs…)
}
```

Why Not Return / Export Channels?

```go
for {
    select {
    case <-ctx.Done():
        return ctx.Err()
    case msg := <-consumer.Messages():
        var event HTTPEvent
        err := json.Unmarshal(msg.Value, &event)
        if err != nil {
            // handle err
        }
        err := processEvent(event)
        // handle err
    case err := <-consumer.Errors():
        // handle err
    }
}
```

If there's one thing you take away from this talk, I implore you to think twice before exporting or returning a channel from your API.
When returning a channel, who's responsible for allocating it? closing it? Is it buffered or unbuffered? Perhaps most of all, how are errors handled effectively? No easy way to link them. Getting errors on another channel complicates error handling.
It's easier to take a blocking API and turn it into a nonblocking one with goroutines, but harder to convert an API exporting channels into one that appears blocking.
Makes middleware / code reuse difficult.

Handlers

```go
type Handler[T1, T2 any] interface {
    Handle(context.Context, Message[T1]) ([]Message[T2], error)
}

type HandlerFunc[T1, T2 any] func(context.Context, Message[T1])
([]Message[T2], error)

func (hf HandlerFunc[T1, T2]) Handle(
  ctx context.Context, msg Message[T1]
) ([]Message[T2], error) {
    return hf(ctx, msg)
}
```

Notice the 2 type parameters.

Handlers can transform, filter, split, do pretty much anything with the event.

```go
func (p *Processor[T1, T2]) handle(ctx context.Context) error {
    for {
        msg, ack, err := p.src.Recv(ctx)
        if err != nil {
            return fmt.Errorf("source: %w", err)
        }
        msgs, err := p.handler.Handle(ctx, msg)
        if err != nil {
            return fmt.Errorf("handler: %w", err)
        }
        if len(msgs) == 0 {
            Ack(ack)
            continue
        }
        err = p.dst.Send(sctx, ack, msgs...)
        if err != nil {
            return fmt.Errorf("destination: %w", err)
        }
    }
}
```

Redis

```go
func (d *Destination) Send(
  ctx context.Context,
  ack func(),
  msgs …kawa.Message[[]byte],
) error {
    // Send to redis server using XADD (publish)
    for _, msg := range msgs {
        _, err := d.client.XAdd(ctx, &goredis.XAddArgs{
            Stream: d.opts.topic,
            Values: map[string]any{"msg": binMarshaler(msg.Value)},
        }).Result()
        if err != nil {
            return err
        }
    }
    kawa.Ack(ack)
    return nil
}
```

Redis

```go
func (s *Source) Recv(ctx context.Context)
  (kawa.Message[[]byte], func(), error) {
    select {
    case <-ctx.Done():
        return kawa.Message[[]byte]{}, nil, ctx.Err()
    case msgAck := <-s.msgC:
        return msgAck.Msg, msgAck.Ack, nil
    }
}
```

```go
func (s *Source) recvLoop(ctx context.Context) (err error) {
outer:
    for {
        msgs, _ := s.client.XReadGroup(ctx, &goredis.XReadGroupArgs{ /* ... */ }).Result()
        for _, msg := range msgs {
            for _, m := range msg.Messages {
                ack := func() {
                    s.client.XAck(ctx, msg.Stream, "kawa", m.ID).Result()
                }
                // ... `bts` unwrapping
                msgAck := kawa.MsgAck[[]byte]{
                    Msg: kawa.Message[[]byte]{
                        Key:   m.ID,
                        Topic: msg.Stream,
                        Value: bts,
                    },
                    Ack: ack,
                }
                select {
                case <-ctx.Done():
                    err = ctx.Err()
                    break outer
                case s.msgC <- msgAck:
                }
            }
        }
    }
    return err
}
```

Batching at the edges.

In the XReadGroup call here, we can decide how many messages we're going to request.

It's recommended for this to be a big enough number to satisfy the work that the processor will do and that the destination can keep up with, but it's configurable!

Concurrency in action. This is fetching new events concurrently with processing and sending to destinations.

```go
type Flusher[T any] interface {
    Flush(context.Context, []kawa.Message[T]) error
}

type FlushFunc[T any] func(context.Context, []kawa.Message[T]) error

// …

batcher := batch.NewDestination[[]byte](
        batch.FlushFunc[[]byte](ret.Flush),
        batch.Raise[[]byte](),
        batch.FlushLength(ret.batchSize),
        batch.FlushFrequency(5*time.Second),
    )
wg.Add(batcher)
```

Batching Semantics

Give the batcher a synchronous flush function
Once that function returns successfully, all the messages in the batch are automatically acknowledged.
Configurable error handling and batching parameters (batch size and timeout)

```go
type JSONFolk struct {
    wrapped kawa.Source[[]byte]
}

func (s JSONFolk) Recv(ctx context.Context) (
   kawa.Message[Folk],
   func(),
   error
 ) {
    data, ack, err := s.wrapped.Recv(ctx)
    if err != nil {
        return kawa.Message[Folk]{}, nil, err
    }
    var folk Folk
    if err := json.Unmarshal(data.Value, &folk); err != nil {
        return kawa.Message[Folk]{}, nil, err
    }
    return kawa.Message[Folk]{Value: folk}, ack, nil
}
```

Middlewares — Serialization is a big one
Can also be used for:

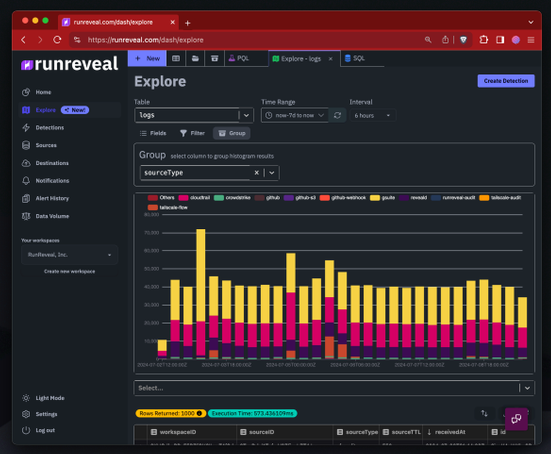
- Metrics
- Routing
- Authentication
- Caching
- ???

Learning from the logs

- Exposing channels across package boundaries introduces questions over channel semantics that need to be handled
- Go generics are great for serialization use cases
- Synchronous / Blocking APIs make it easier to implement middleware and allow for broader code-reuse
- Batch at the edges, process events serially internally

Batching at the edges is an important takeaway too. Tcp congestion control, io_uring, kafka consumers, all batch internally but expose APIs that look like one event at a time.

runreveal



https://www.gokawa.dev

https://runreveal.com

If this sounds interesting,
Come talk to me!

I have Stickers!