
AppDeveloperCon
NORTH AMERICA

AppDeveloperCon NORTH AMERICA

Applying CloudEvents to Level Up Application Communications

Vyom Yadav, Canonical
Evan Anderson, Stacklok

Applying CloudEvents to Level Up Application Communications

AppDeveloperCon
NORTH AMERICA

November 12, 2024
Salt Lake City

Vyom Yadav
Software Engineer
Canonical

Evan Anderson
Principal Software Engineer
Stacklok

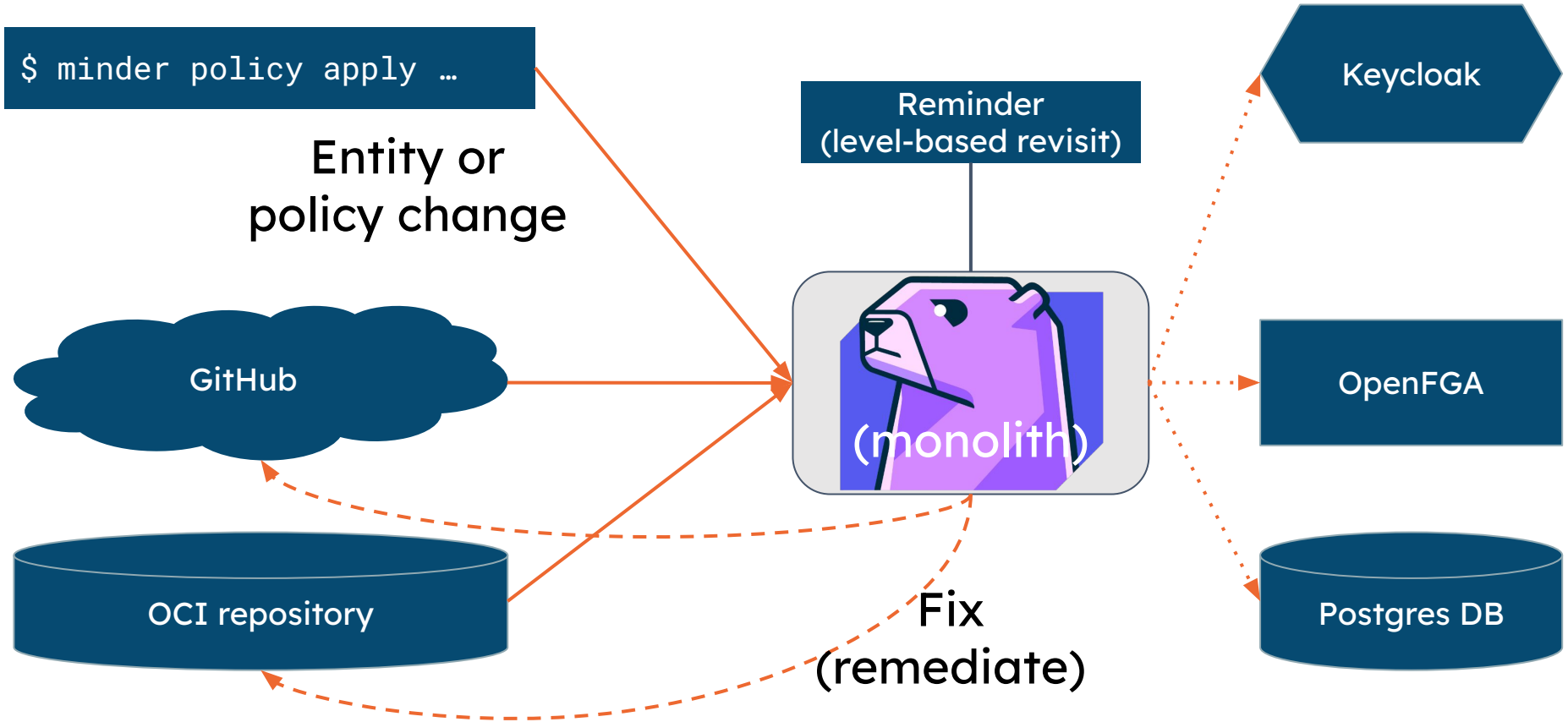
minder

AppDeveloperCon NORTH AMERICA

> A platform that helps teams build more secure software, and prove to others that what they've built is secure. Minder helps project owners proactively manage their security posture by providing a set of checks and policies to minimize risk along the software supply chain.

(An OpenSSF project)

Architecture

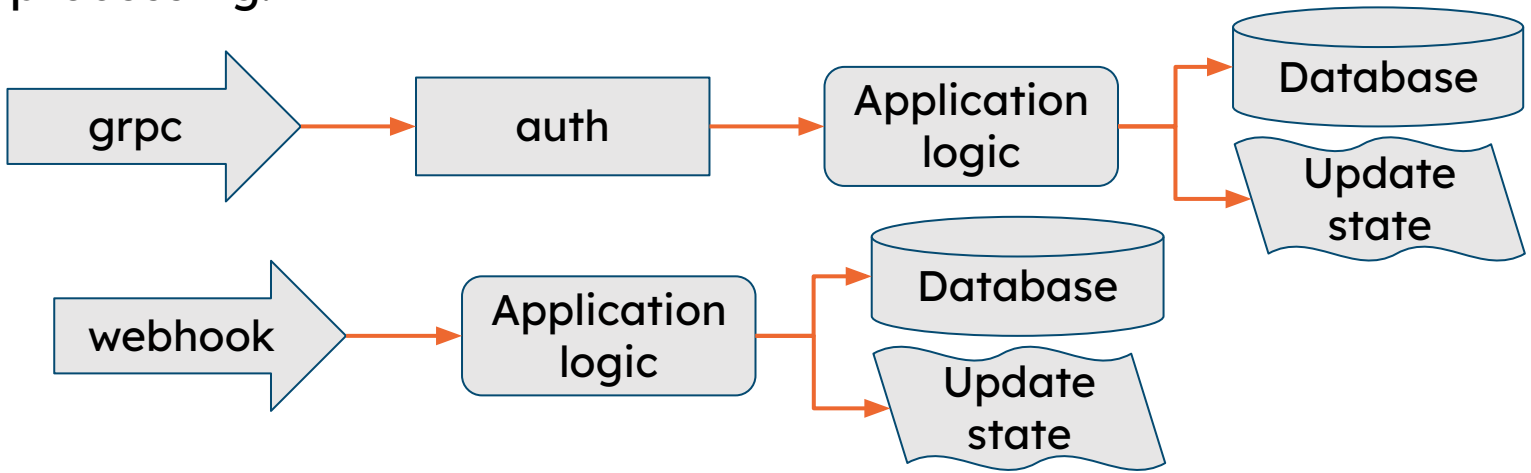


First Architecture

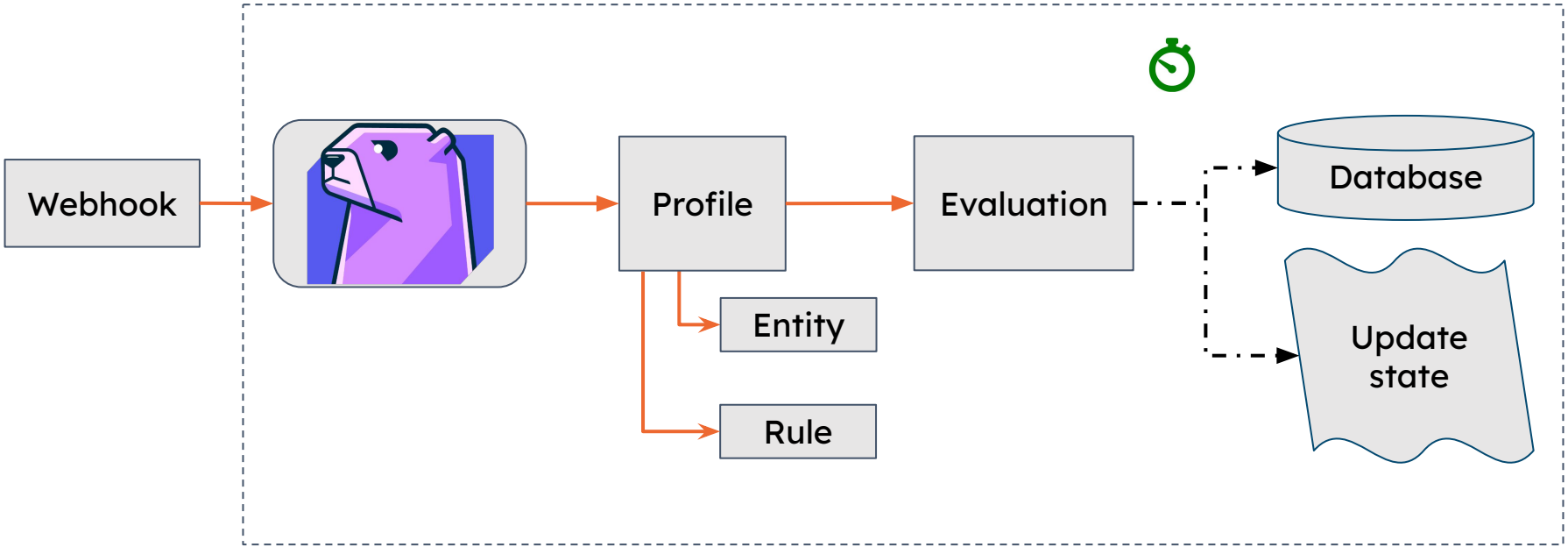
Lots of input sources:

- Command-line: GRPC (+GRPC-gateway)
- GitHub: webhooks (`google/go-github` + custom code)
- OCI registry: more webhooks, mostly

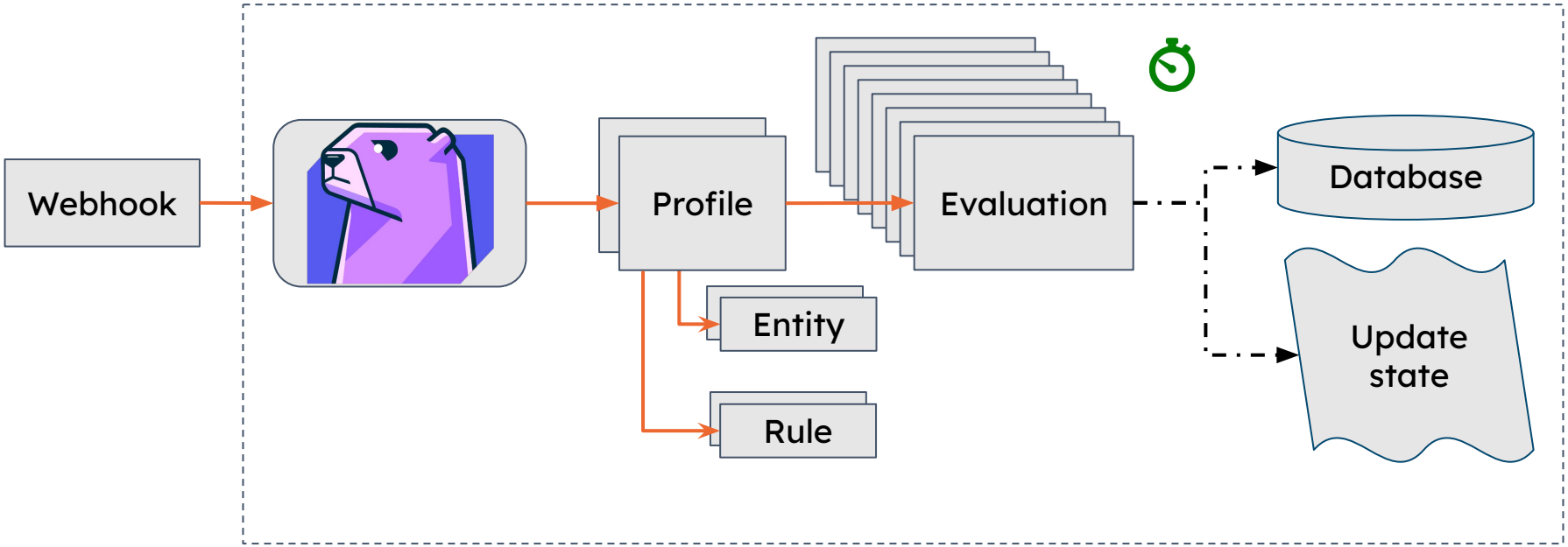
Input processing:



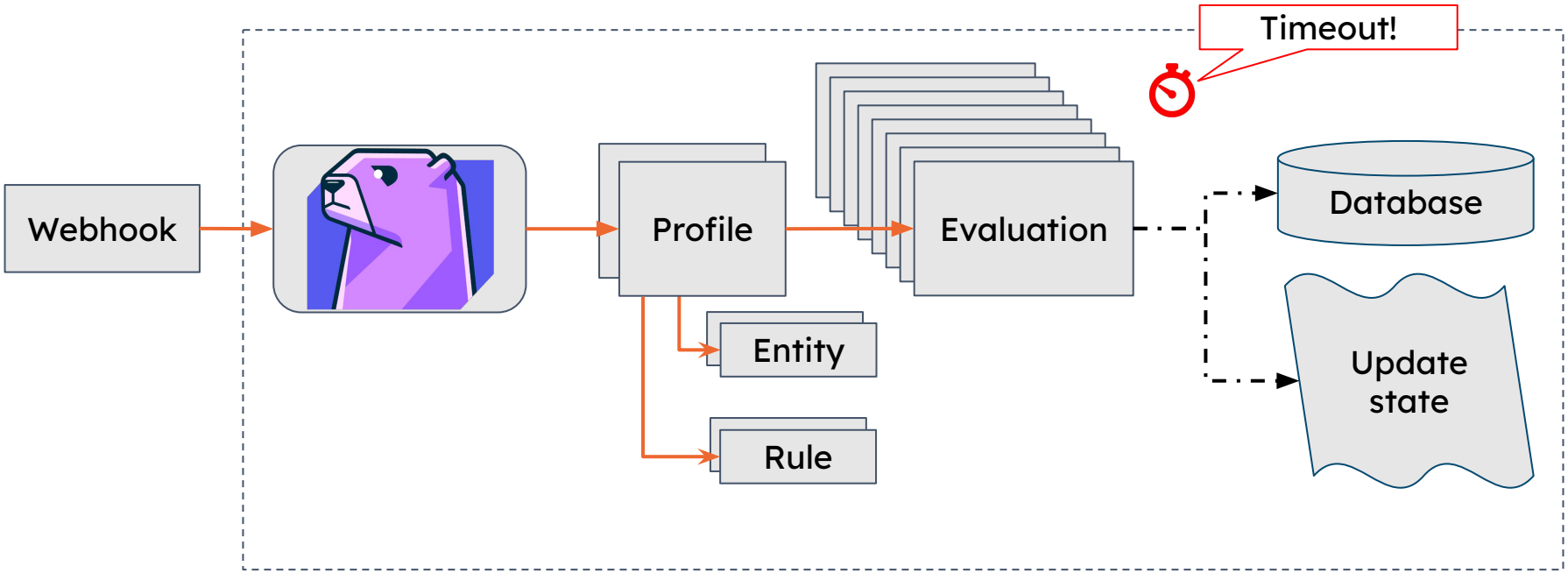
Problems in Minder-land



Problems in Minder-land



Problems in Minder-land



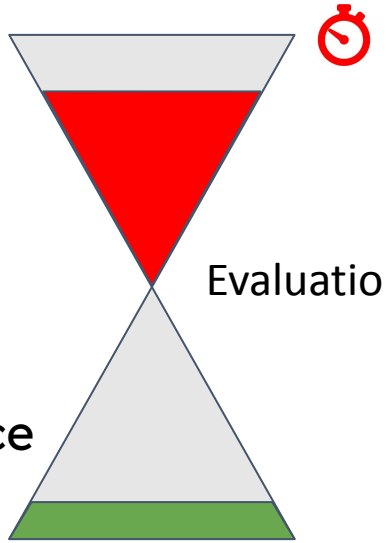
The Culprit: Profile Evaluation

Whenever an entity or rule would change, we needed to evaluate a bunch of policies.

Where a policy might look like:

- Call this GitHub API and check the results
- Clone a GitHub repo, and check if a file is present
- Clone a GitHub repo, and check file contents
- Check files in a pull request against an external service
- Check the signatures on an artifact

Doing a lot of this was… a lot of work!



Asynchronous Requests to the Rescue?

Two common patterns for application communication:

Synchronous

- Make a request, wait for a reply

Asynchronous

- Send a message which gets delivered later

Synchronous Messaging

Simplest, most common type of communication.

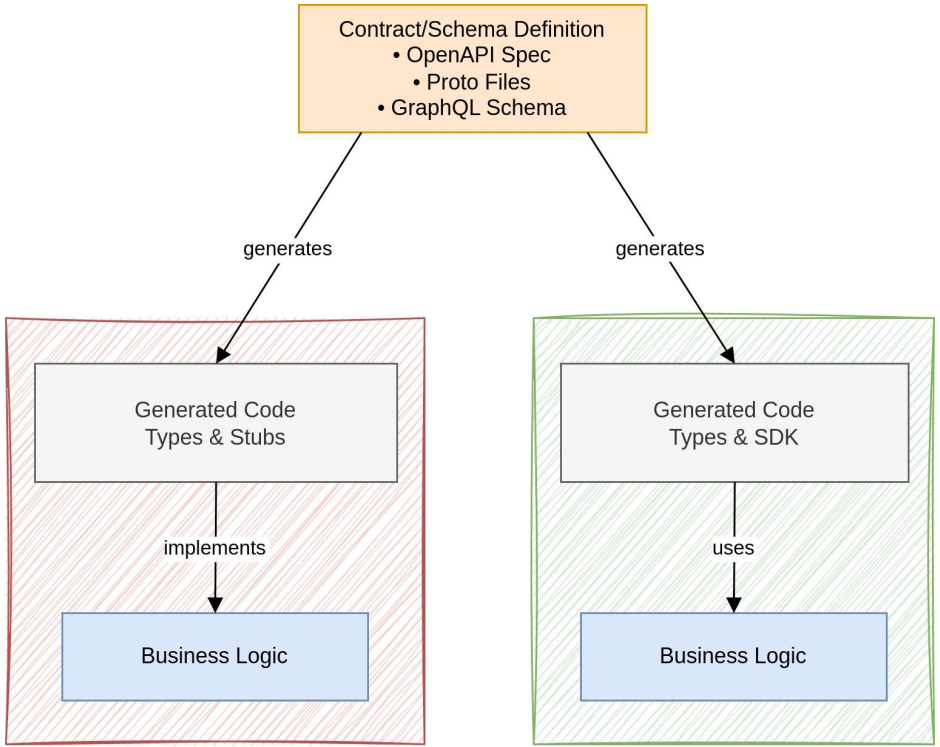
👍 Care about getting an answer

👍 Request completes before proceeding

Challenges:

- time-bounded (client can time out waiting for server)
- single destination (only send to one server at a time)

Synchronous Ecosystem Standardization



OPENAPI

gRPC

GraphQL

Asynchronous Messaging

For work that can happen sometime later.

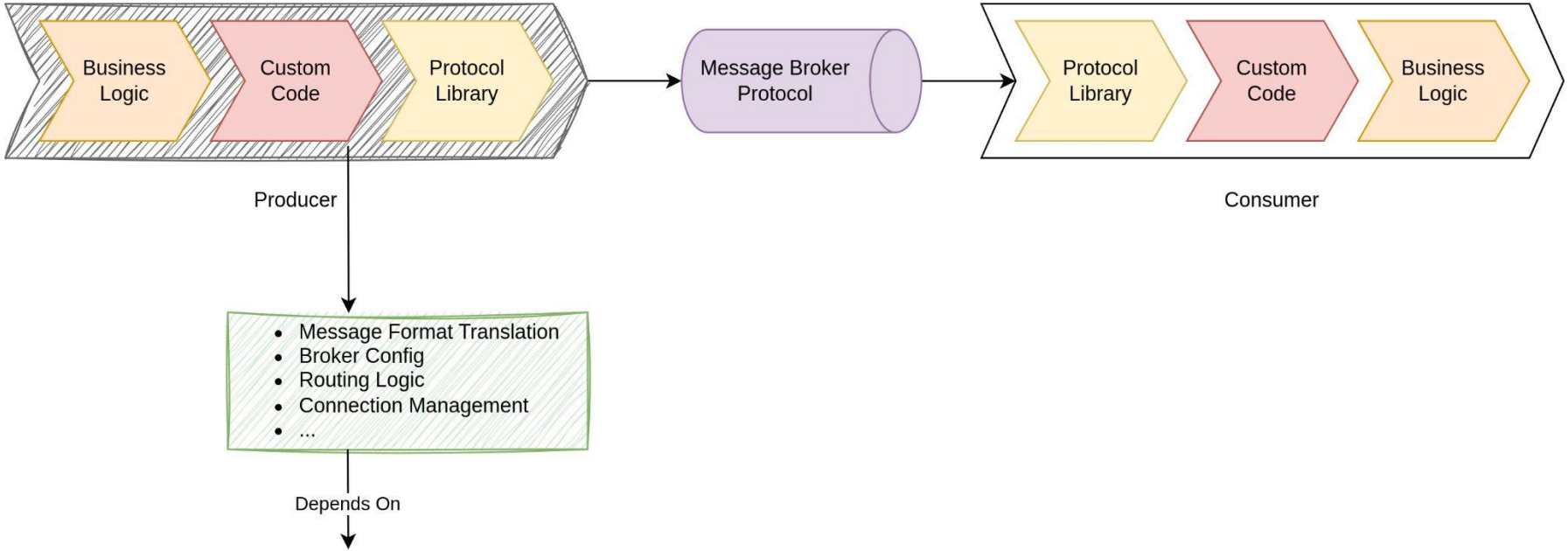
👍 One message can be routed to multiple recipients (if desired)

👍 Don’t have to block until the work is done

Challenges:

- original sender can’t easily get results
- error detection and handling – need retries and dead letter queues

Asynchronous Ecosystem ~~Standardization~~



+ Asynchronous is already complicated enough.
Write business logic, not custom code.

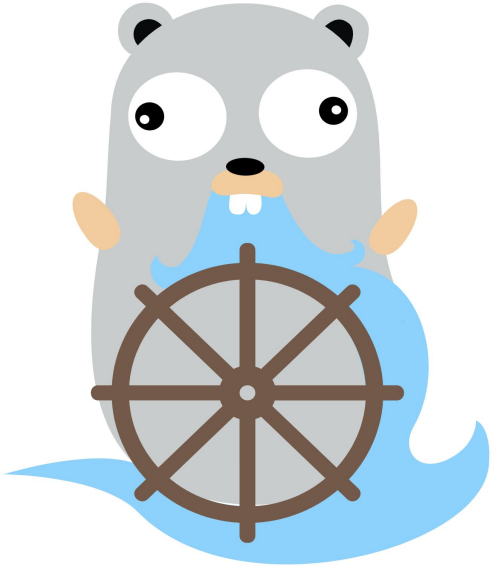
Watermill

Go-native event processing library

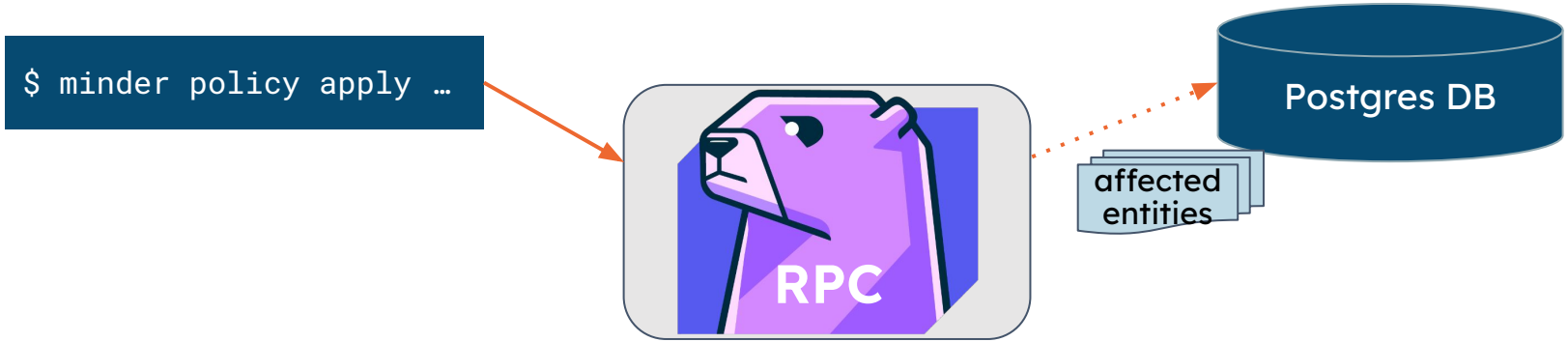
Hit a number of our needs:

- Easy to get started with `GoChannel` for testing
- Supports several transports with persistence
- Designed for at-least-once delivery (retries)
- Supports filters and fan-out

Decent documentation: watermill.io/docs

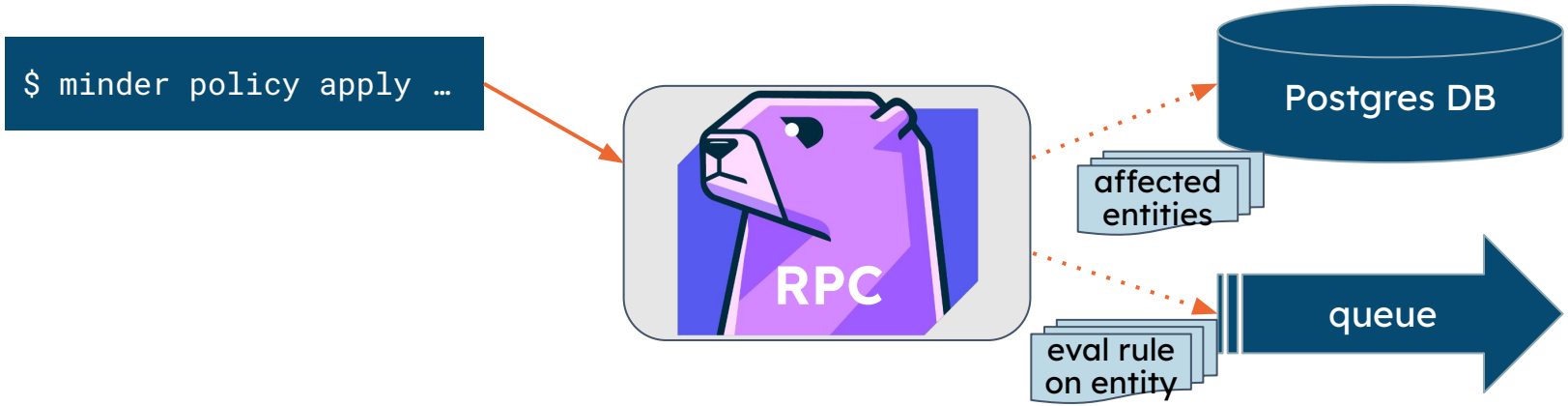


Minder Workflow With Events



When a new policy is applied or a repository is registered, Minder evaluates which entities + policies need to be applied.

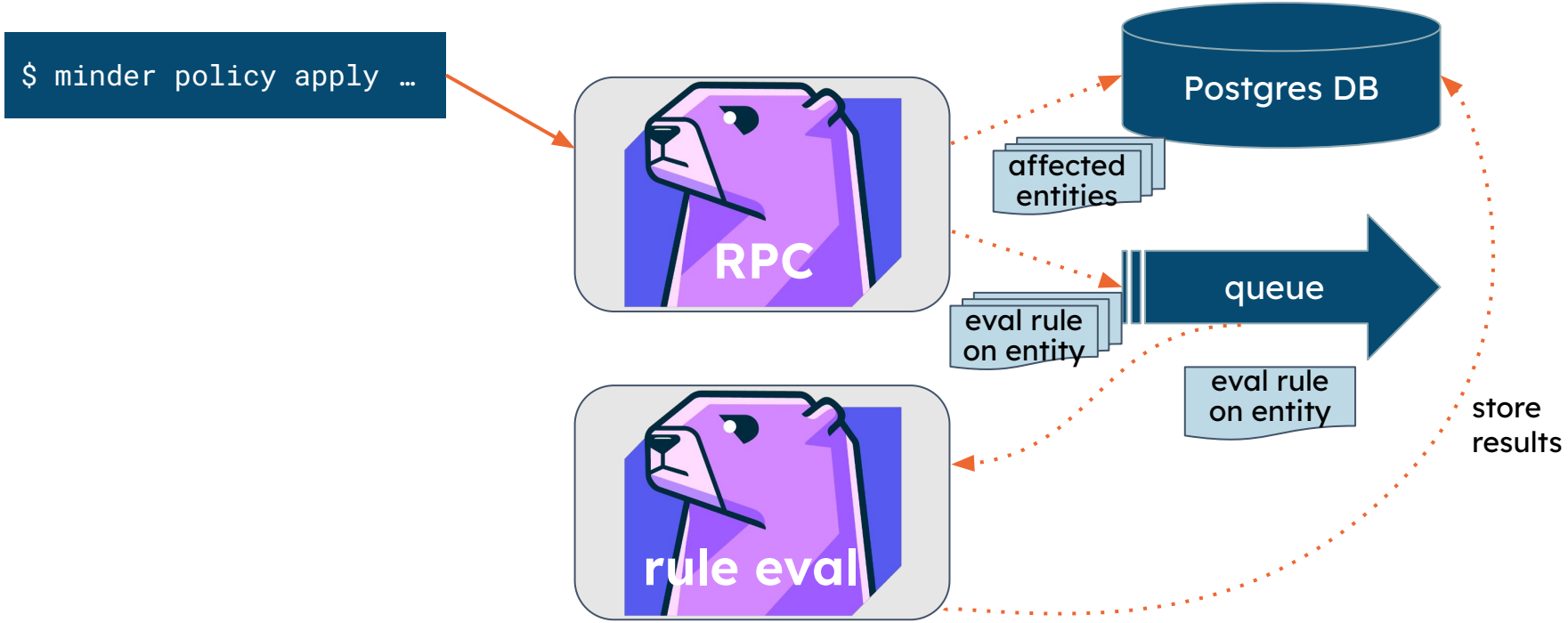
Minder Workflow With Events



For each policy & entity combination, we generate an event which indicates that the policy needs to be re-evaluated.

Note: these events don't describe the change, just the need to re-evaluate. Minder is level-triggered, like Kubernetes!

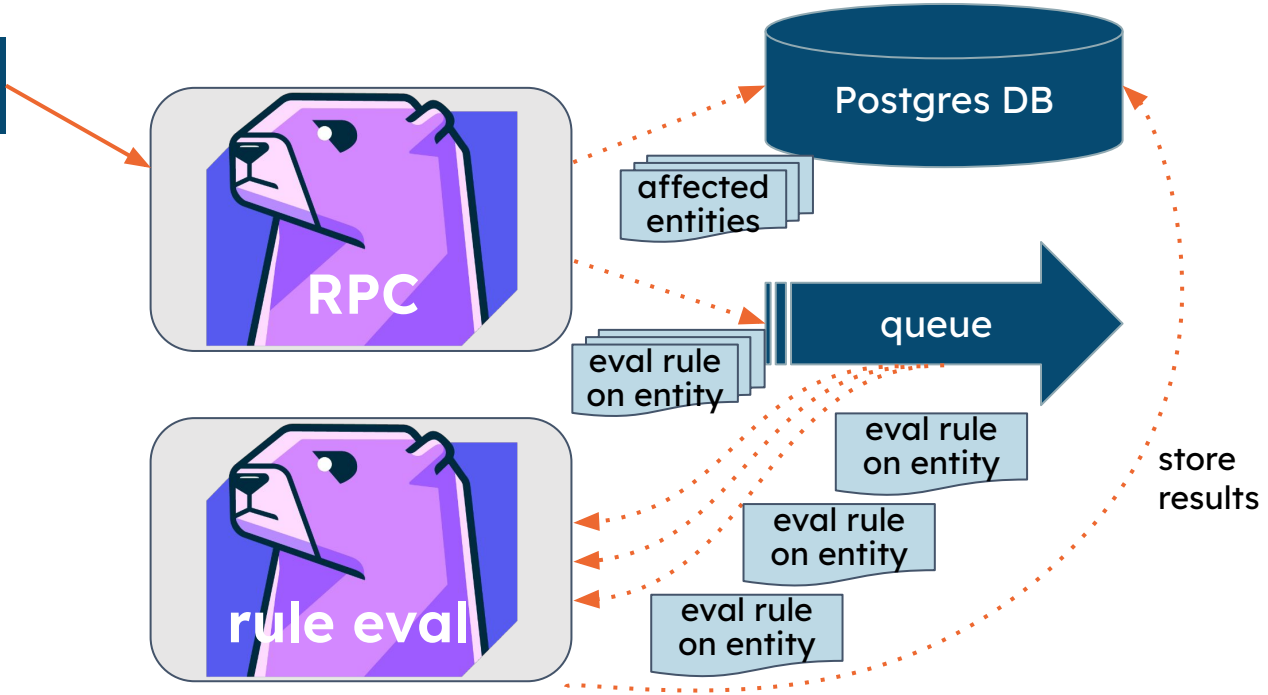
Minder Workflow With Events



Minder's rule evaluation runs asynchronously, with the results stored back in the database in an evaluation history table.

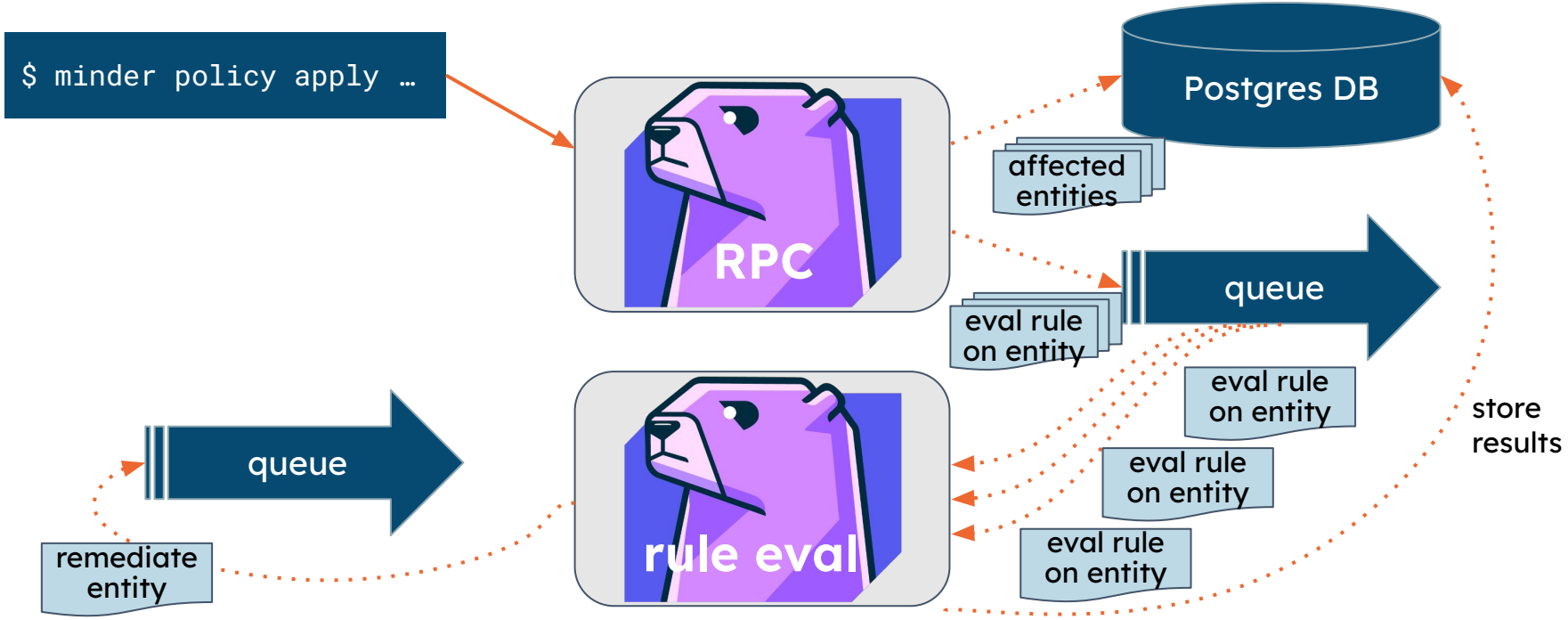
Minder Workflow With Events


```
$ minder policy apply …
```



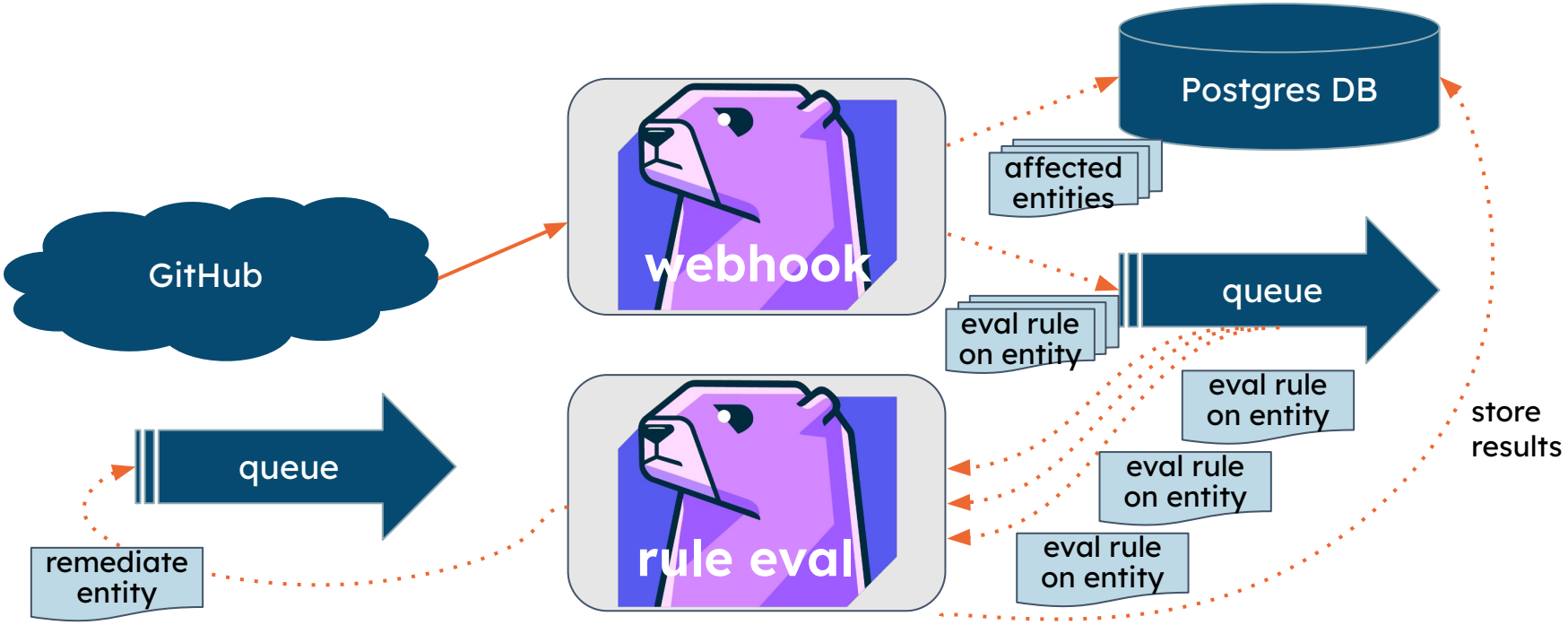
When a policy is applied to a large number of repos, we can use the queue to limit the number of rule entity evaluations in flight.

Minder Workflow With Events



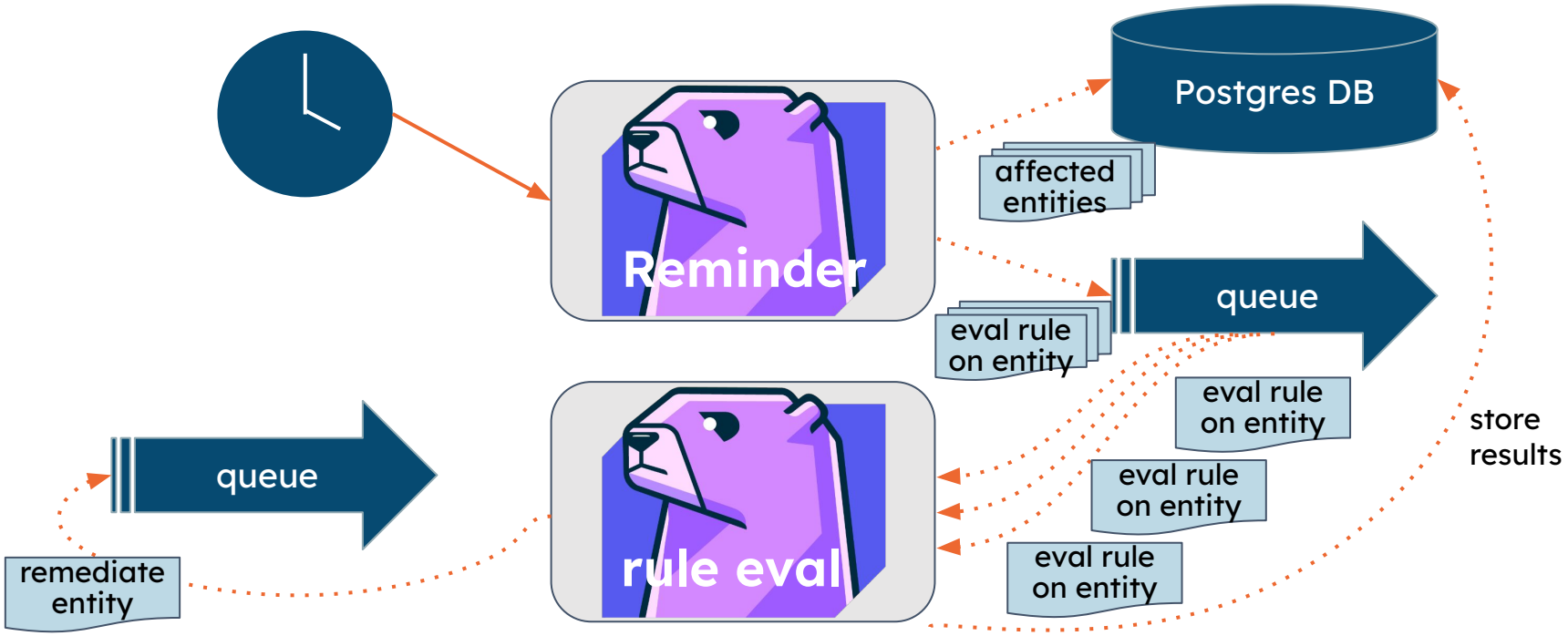
We can also use a queue to dispatch remediations when needed from rule evaluation.

Minder Workflow With Events



GitHub change notifications (via webhook) are handled in the same way as RPCs – while one repo is changing, it may trigger many policy evaluations.

Minder Workflow With Events



Reminder is a secondary process to ensure level-based processing of entities. It periodically re-visits entities and sends a rule eval event.

Minder Async Use Cases

Webhook notifications

- Entity changed, re-scan and re-apply policy
- “Kick” all the rules (level based)

Rule / Profile update

- If the rule definition changes, revisit all entities the rule applies to

Periodic revisit (level trigger)

- We might miss an edge: visit untouched entities periodically
- Like the Kubernetes `resyncPeriod` on informers

Account / Project delete

- Cascading delete has a lot of work that’s not urgent

Watermill, 12 months in

Good:

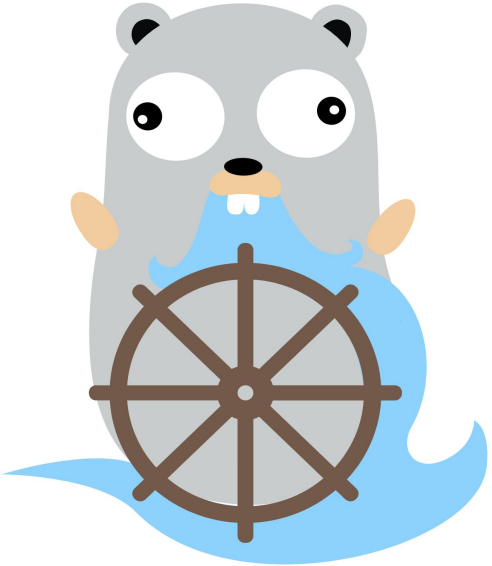
- Message durability
- Postgres implementation was easy to set up

Bad:

- Go-specific
- Postgres implementation performance is so-so.
- `Message` doesn't have standard metadata like event time, tracing, customer…

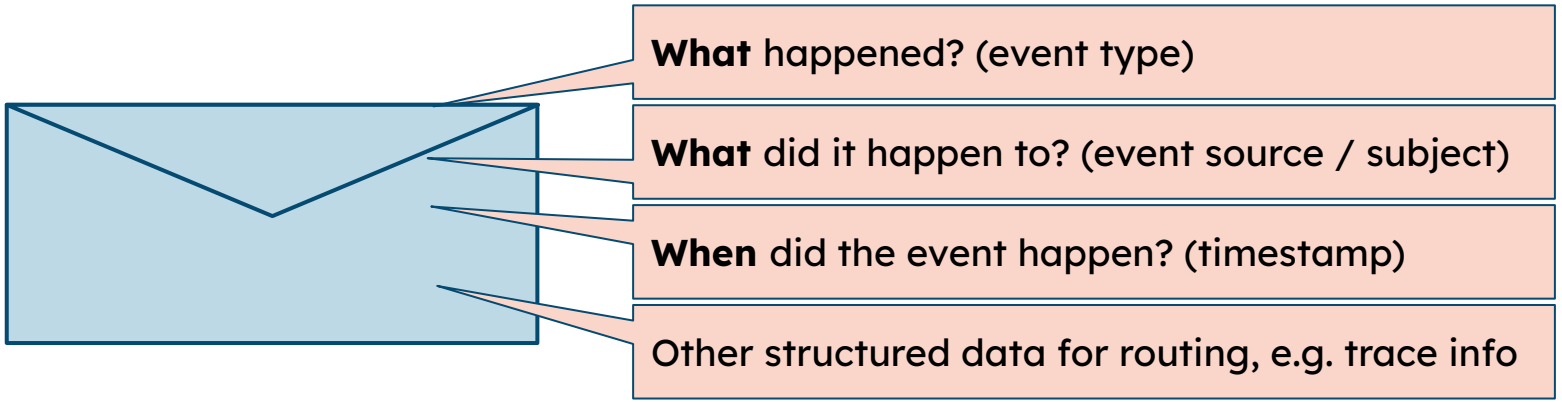
Ugly:

- Postgres channel is very expensive
- Poison / Dead Letter middleware sometimes gets retried infinitely due to layering



CNCF Cloud Events

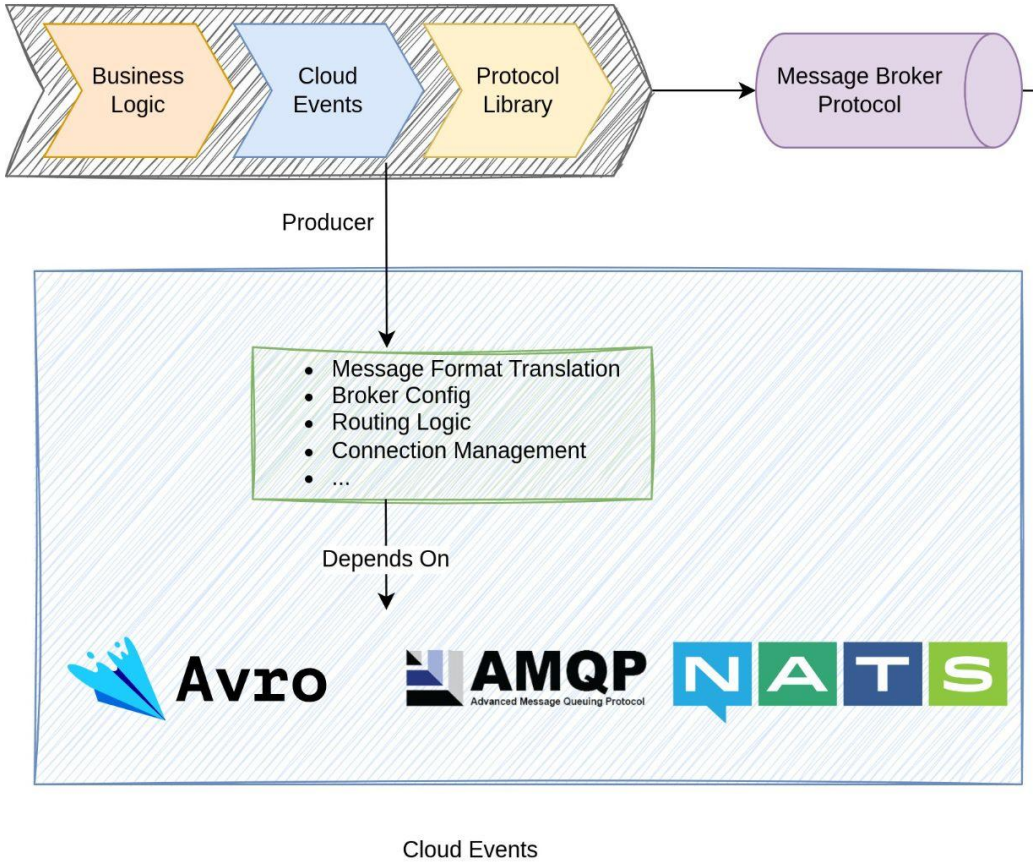
Provides a standardized *envelope* for asynchronous messages.



This envelope is defined for multiple *transports*:

AMQP Advanced Message Queuing Protocol · APACHE kafka · MQTT · NATS · HTTP · ● ● ●

Asynchronous Ecosystem Standardization



CloudEvents Library:

- Metrics and tracing collection
- Correlation IDs
- Routing and selection

Transport Library:

- Storage and routing
- Retries / Dead-letter
- Routing and selection

Cloud Events introduces standardization and removes the need to write custom code.

What's Different?

Watermill abstracts all the event processing

- Each `Message` has its own `Context`
- Publishers and Subscribers are configured within Watermill config

CloudEvents is an envelope that works with the native transport library

```
publisher, err := nats.NewPublisher(
     nats.PublisherConfig{
          URL:         natsURL,
          NatsOptions: options,
          Marshaler:   marshaler,
          JetStream:   jsConfig,
     },
     logger,
)
if err != nil {
     return nil, err
}
```

```
consumer, err := ce_nats.NewProtocol(
     c.cfg.URL, c.cfg.Prefix, topic, topic,
     opts, jetstreamOpts, subOpts, …))
if err != nil {
     return nil, err
}


ceSub, err := ce.NewClient(consumer, …)
```

How we're doing it

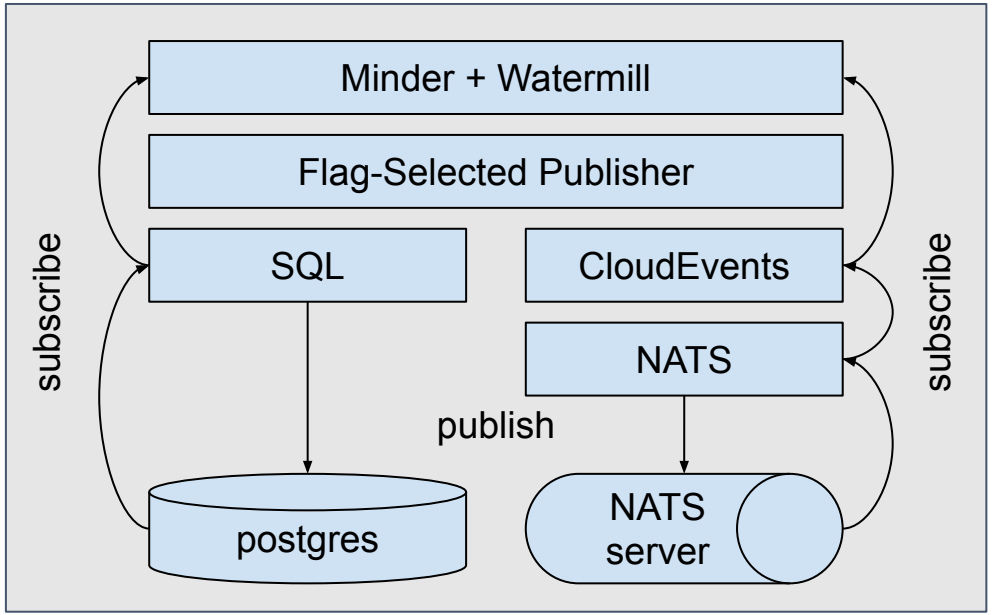
Migrating message queues is tricky:

- Don't want to lose messages
- Want to put weight on new system
- Avoid a "point of no return"

Approach:

- Implement new publisher
- Listen to old & new subscribers
- Publish *some* messages to the new publisher

Simplify *after* migration



Takeaways

CloudEvents is especially handy if you're multi-language or multi-transport

- Particularly easy to route from one transport to another

Specify your asynchronous flows as CloudEvents

- type, source, subject, payload, and any custom attributes

Consider using CloudEvents as an *external* API for users

- For example – we could have exposed the history of reconciliations as CloudEvents

Additional Practical Advice

CloudEvents won't replace your full async stack. It's another layer on top of your transport library.

Whether or not you use CloudEvents, you probably want to encapsulate your async messaging behind your *own* `Publish` and `Subscribe` abstractions for later portability.

At least a few CNCF projects to help with this: