reliability nirvana

AKA Event Driven Design

Daniel Selans // daniel@batch.sh

Disclaimer

- Everything I talk about, I have done in production
  - .. and if I haven't, I'll tell you
- No fluff
  - Some stuff is hard, some stuff is easy - will keep it real!
- My personal goal
  - You have **leveled up**

Obligatory `whoami`

- Reside in Portland, OR
- Working in backend for 10+ years
- Love building and operating distributed systems
- Previously
  - R&D @ data centers
  - SRE @ New Relic
  - SE @ InVisionApp
  - SE @ DigitalOcean
  - SE @ Community
- And most recently - co-founder/CTO at **Batch.sh**
  - Observability platform for high throughput data
  - Got into YCombinator (S20) - woohoo! 🎉

Obligatory `whoami`

- Reside in Portland, OR
- Working in backend for 10+ years
- Love building and operating distributed systems
- Previously
  - R&D @ data centers
  - SRE @ New Relic
  - SE @ InVisionApp
  - SE @ DigitalOcean
  - SE @ Community
- And most recently - co-founder/CTO at **Batch.sh**
  - Observability platform for high throughput data
  - Got into YCombinator (S20) - woohoo! 🎉

Oh, and I'm from 🇱🇻 !

What is “Reliability Nirvana”?

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night
- **Predictable *service* failure scenarios**

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night
- Predictable *service* failure scenarios
- **Well-defined *service* boundaries**

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night
- Predictable *service* failure scenarios
- Well-defined *service* boundaries
- **Security conscious**

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night
- Predictable *service* failure scenarios
- Well-defined *service* boundaries
- Security conscious
- **Self-healing services**

“Reliability Nirvana” is...

- Not being woken up at 3AM on a Saturday night
- Predictable *service* failure scenarios
- Well-defined *service* boundaries
- Security conscious
- Self-healing
- **Highly scalable and highly reliable**

“But we already have $insert_tech...”

“But we already have $insert_tech...”

- **Microservices pattern**
  - Tackles the monolith problem

“But we already have $insert_tech...”

- Microservices pattern
  - Tackles the monolith problem
- **Containers (docker)**
  - Tackles reproducible builds/releases, dev flow

“But we already have $insert_tech...”

- Microservices pattern
  - Tackles the monolith problem
- Containers (docker)
  - Tackles reproducible builds/releases, dev flow
- **Container orchestration (kubernetes, mesos, rancher)**
  - Tackles container lifecycle

“But we already have $insert_tech...”

- Microservices pattern
  - Tackles the monolith problem
- Containers (docker)
  - Tackles reproducible builds/releases, dev flow
- Container orchestration (kubernetes, mesos, rancher)
  - Tackles container lifecycle
- **Service mesh (linkerd, istio, consul)**
  - Tackle inter-service communication

“But we already have $insert_tech...”

- Microservices pattern
  - Tackles the monolith problem
- Containers (docker)
  - Tackles reproducible builds/releases, dev flow
- Container orchestration (kubernetes, mesos, rancher)
  - Tackles container lifecycle
- Service mesh (linkerd, istio, consul)
  - Tackle inter-service communication
- ***And probably several other quality of life improvements***

Things sound pretty good. What’s the problem?

Achieving *really* high reliability is **hard**

- Lots of inter-dependent microservices == huge failure domain
  - 1 service being slow will result in unpredictable system state
- So.. use histrix-style circuit breakers!
  - .. turns out, it's pretty hard to think of all possible failure scenarios
  - … and it's easy to shoot yourself in the foot
- .. And avoid cascading failures!
  - "Well, service G knows how to deal with that situation and it won't happen"
- .. And we need self-healing at service level!
  - Will mid-flight requests survive auto-scale events?
- .. And keep security in mind!
  - Because your PM loves it when you don't ship features! 😅

I sense a pattern...

- Services **do not** have to rely on each other
- Services are able to **easily** recover from where they were at when they failed or were restarted
- Developers **do not** have to implement complex, per-service circuit breaker & fault-tolerance strategies
- SRE's **do not** have to define per-service firewall rules
- Be able to go through *every* state change in a req/tx
- Expose all of your backend data for future analytics uses

Reliability Nirvana is …

Effortless service reliability

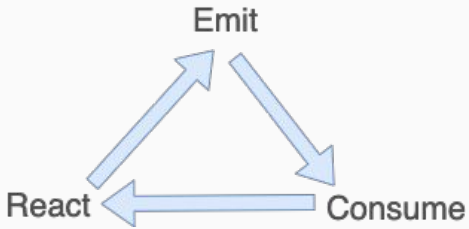
Event Driven

reliability nirvana

Software (*and systems*) architecture paradigm promoting the: **production**, **detection**, **consumption** of, and **reaction** to events.

Core concepts for Event Driven

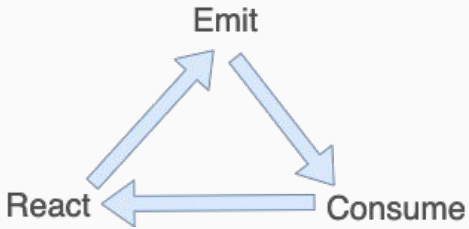
- At its core, event driven consists of three actions:



- **Events MUST be the source of truth**
  - A single service doesn't know the state of the ***system*** - it only knows its ***OWN*** state

Core concepts for Event Driven

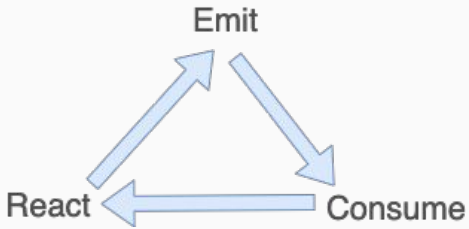
- At its core, event driven consists of three actions:



- Events MUST be the source of truth
  - A single service doesn't know the state of the ***system*** - it only knows its ***OWN*** state
- **All events are communicated through an event/message bus**

Core concepts for Event Driven

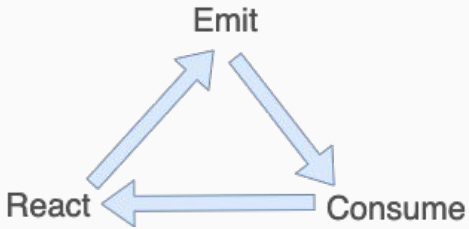
- At its core, event driven consists of three actions:



- Events MUST be the source of truth
  - A single service doesn't know the state of the ***system*** - it only knows its ***OWN*** state
- All events are communicated through an event/message bus
- **All services MUST be idempotent**
  - Your services are able to work with duplicate events
  - Your services are able to work with out-of-order events

Core concepts for Event Driven

- At its core, event driven consists of three actions:



- Events MUST be the source of truth
  - A single service doesn't know the state of the ***system*** - it only knows its ***OWN*** state
- All events are communicated through an event/message bus
- All services MUST be idempotent
  - Your services are able to work with duplicate events
  - Your services are able to work with out-of-order events
- **You MUST be OK with eventual consistency**
  - You *trust* that the system will be eventually consistent but cannot guarantee consistency

Event Driven Components

reliability nirvana

Event Driven Components

- **Event bus**
  - All events pass through here as protobuf

- You want **RabbitMQ**
  - #1 reason - versatile routing configuration
  - Medium-fast (~20k msgs/s)
  - Reliable, little to no babysitting

Event Driven Components

- Event bus
  - All events pass through here as protobuf
- **Caching / config layer**
  - As your service consumes events, store the resulting state here. On restarts, load the state into memory.

- You want **RabbitMQ**
  - #1 reason - versatile routing configuration
  - Medium-fast (~20k msgs/s)
  - Reliable, little to no babysitting
- You want **etcd**
  - Rock solid, high latency resilient
  - Medium-fast (20k+ msgs/s)
  - Reliable, little to no babysitting

Event Driven Components

- Event bus
  - All events pass through here as protobuf
- Caching / config layer
  - As your service consumes events, store the resulting state here. On restarts, load the state into memory.
- **Long term event storage**
  - All of the events that ever pass through the message bus should be stored here. Forever.

- You want **RabbitMQ**
  - #1 reason - versatile routing configuration
  - Medium-fast (~20k msgs/s)
  - Reliable, little to no babysitting
- You want **etcd**
  - Rock solid, high latency resilient
  - Medium-fast (20k+ msgs/s)
  - Reliable, little to no babysitting
- You want **S3**
  - Cheap, fast, reliable

Event Driven Components

- Event bus
  - All events pass through here as protobuf
- Caching / config layer
  - As your service consumes events, store the resulting state here. On restarts, load the state into memory.
- Long term event storage
  - All of the events that ever pass through the message bus should be stored here. Forever.
- **Event archiver**
  - A custom service you will build to consume events from the message bus to populate your long term store

- You want **RabbitMQ**
  - #1 reason - versatile routing configuration
  - Medium-fast (~20k msgs/s)
  - Reliable, little to no babysitting
- You want **etcd**
  - Rock solid, high latency resilient
  - Medium-fast (20k+ msgs/s)
  - Reliable, little to no babysitting
- You want **S3**
  - Cheap, fast, reliable
- You want **Go**
  - Easy to write performant code
  - Great libs

How does this translate to improved reliability?

- **You do not have to think about service outages**
  - The service will **eventually** read all of the messages it may have missed

How does this translate to improved reliability?

- You do not have to think about service outages
  - The service will **eventually** read all of the messages it may have missed
- **You have a MUCH smaller failure domain**
  - Services do not depend on each other - no cascading failures
  - Predictable failure mode
  - 1 service outage == 1 feature outage

How does this translate to improved reliability?

- You do not have to think about service outages
  - The service will **eventually** read all of the messages it may have missed
- You have a MUCH smaller failure domain
  - Services do not depend on each other - no cascading failures
  - Predictable failure mode
  - 1 service outage == 1 feature outage
- **True service autonomy**
  - Teams no longer have to “depend” on another service/team

How does this translate to improved reliability?

- You do not have to think about service outages
  - The service will **eventually** read all of the messages it may have missed
- You have a MUCH smaller failure domain
  - Services do not depend on each other - no cascading failures
  - Predictable failure mode
  - 1 service outage == 1 feature outage
- True service autonomy
  - Teams no longer have to “depend” on another service/team
- **Well-defined development workflow (via protobuf schemas)**
  - Protobuf schema clearly defines what message(s) you should send or receive

How does this translate to improved reliability?

- You do not have to think about service outages
  - The service will **eventually** read all of the messages it may have missed
- You have a MUCH smaller failure domain
  - Services do not depend on each other - no cascading failures
  - Predictable failure mode
  - 1 service outage == 1 feature outage
- True service autonomy
  - Teams no longer have to “depend” on another service/team
- Well-defined development workflow (via protobuf schemas)
  - Protobuf schema clearly defines what message(s) you should send or receive
- **Dramatically lower attack surface!**
  - Services no longer have to talk to each other

Event Driven Code Example

https://github.com/batchcorp/go-template

This sounds complex...

reliability nirvana

Yep, it's complicated.

- Requires **excellent** understanding of your message bus tech
- Requires **everyone** to be onboard
  - Create docs, flows, examples, etc.
- Accept that the event bus is your **source of truth**
- Embrace eventual consistency
- Embrace idempotency
- Anticipate complex debug

Implementation Reality: Technical

- Easy: Setup foundational infrastructure (<2 weeks)
  - Event bus, cache, event storage
- Medium: Defining schemas (<1 week)
  - Define message envelope, encoding type (protobuf? Avro? JSON schema?)
- Medium/Hard: Setting up schema publishing/consumption pipeline (~1 week)
  - Compile PB's on update, publish latest pkg
- Medium/Hard: Provide example service that uses event driven (~2 weeks)
- Hard*: Build event archiving solution (~2-4 weeks)
  - How to group/batch events, maximize storage efficiency (and retain ease of use)?
- Hard*: Build a replay mechanism (3+ weeks)
  - How do you efficiently read from the event store?
- Hard*: Build event viewer/search (4+ weeks)
  - How do you search your event store?

* == Do you need it right away?

Implementation Reality: Tips

- Brand new org
  - The most freedom you will ever have - can create a beautiful foundation
  - But…
    - Only implement if you have the full picture
    - Only implement if you are confident in your engineering capability
    - Should have at least a few principal-level engineers with architecture experience
  - Do NOT use CDC (change data capture) as your source of truth (unless you have a *very* good reason)
- Existing org
  - Move to event driven *gradually*
  - Do NOT attempt a move in one fell swoop - it <u>will</u> fail!
    - Functionality will be missed
    - Engineering is not yet accustomed to operating an event driven arch in production
    - It will take 4x as long as you think it will take (and it still won't be complete)
  - Do a "soft-intro" to event driven by utilizing CDC (change data capture)
    - Capture all INSERT/UPDATE/DELETE's and expose them as events

Implementation Reality: Tips

- SRE/platform must ALWAYS be a part of the conversation for distributed system design
  - .. and should usually LEAD the conversation
- If you are not involved:
  - Involve yourself
  - You know best what is or isn't possible on a platform level
- Most of this space is greenfield
  - You *will* have to develop tools
  - Very few tools will fit *exactly* what you're trying to do
- You will have to wear an architecture hat .. whether you like hats or not
- Establish a written culture and get comfortable with writing documentation
  - An event driven system can feel like magic when it "just works"
  - .. but will be daunting to debug when things break - you will want docs and runbooks

In exchange for complexity, you gain:

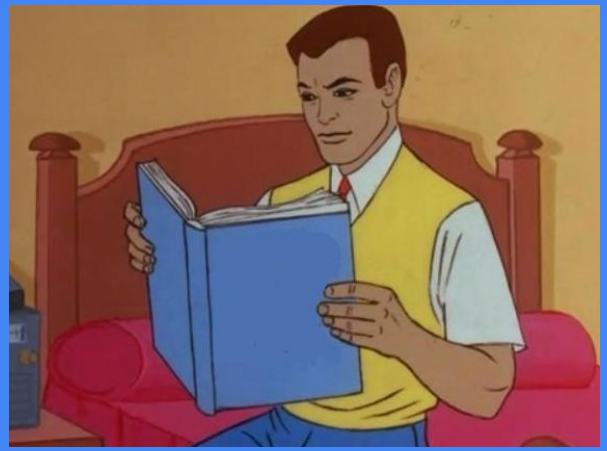
- True service autonomy
- True team autonomy
- Can **always** rebuild state
- Predictable failure scenarios
- Improved outage recovery time
- Ability to sustain long-lasting outages
- Dramatically improved security
- A moldable, robust foundation
- Solid, well-defined architecture
- Lifetime historical records!

Batch.sh uses an event-driven architecture

- AWS EKS (managed k8s), AWS MSK (managed kafka), AWS EC2
- **19** (golang) microservices
- **100%** event driven (except for the frontend <-> public API)
- **0** inter-service dependencies
  - Most services have 3 dependencies - rabbit, etcd and kafka
- No service mesh, no service discovery - **not needed**
- Instead of triggering behavior via curl or postman, we trigger behavior by publishing an event on the bus (using plumber)
- Network is highly locked down
  - Inbound is limited to K8S compute node IP's
  - Outbound is limited to rabbit, kafka and etcd
- Stats
  - Average event size @ 4KB
  - Total ~15M system events, ~100GB storage in S3 (since Dec 2020)

Bonus reading

Here's some additional reading material that you may find useful when diving deeper into event driven.



- **Martin Fowler's "What do you mean 'Event Driven'?"**
  - https://martinfowler.com/articles/201701-event-driven.html
- **Event sourcing**
  - https://microservices.io/patterns/data/event-sourcing.html
- **CQRS**
  - https://martinfowler.com/bliki/CQRS.html
- **Idempotent consumer**
  - https://microservices.io/patterns/communication-style/idempotent-consumer.html
- **Data design for event driven systems**
  - https://www.ben-morris.com/data-design-for-event-driven-architecture-autonomy-encapsulation-and-ordering/
  - "Bear in mind that if you rely on message ordering then you are effectively coupling your applications together in a temporal, or time-based, sense."
- **Exactly-once delivery is … difficult**
  - Avoid at all costs.

Finally...

reliability nirvana

Come talk to me!

Batch.sh is building truly novel stuff.

If you are interested in solving *new* problems and are passionate about distributed systems, let's chat!

Our stack/tech:

- 100% event driven
- K8S
- Golang backend
- Electron; react
- Virtually ALL message brokers
- Etcd
- TimescaleDB
- ElasticSearch
- AWS S3 & Athena
- Multi-cloud

batch

Y

Batch is a data pipeline company that enables high-throughput data observability.

Our platform enables you to:

- Gain visibility into your message bus
  - Expose difficult to "see" data to your devs & data scientists
- Populate your data lake with optimized parquet data
  - With 100% hands-free schema evolution
- Recover from outages by replaying data
- Create a robust backup & disaster recovery strategy
- Improve your tests by using real event data

Shoot me an email: daniel@batch.sh

Ping me on Gophers Slack: Daniel Selans

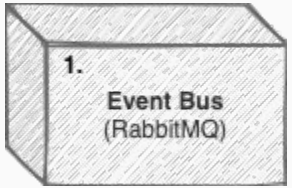
Extra Content

reliability nirvana

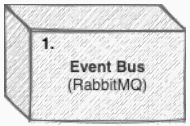
Let’s design an event driven system!

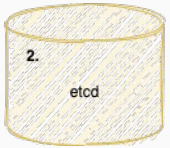
reliability nirvana

1. Event Driven From Scratch

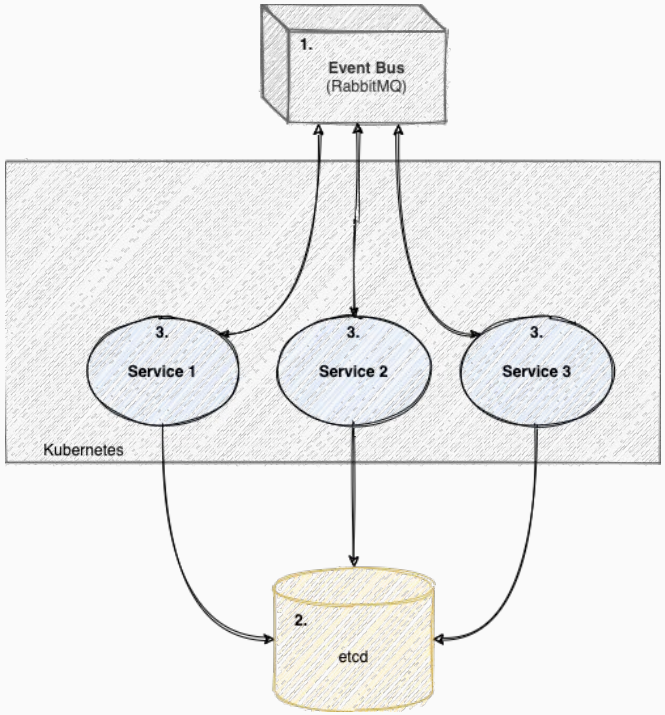


2. Event Driven From Scratch

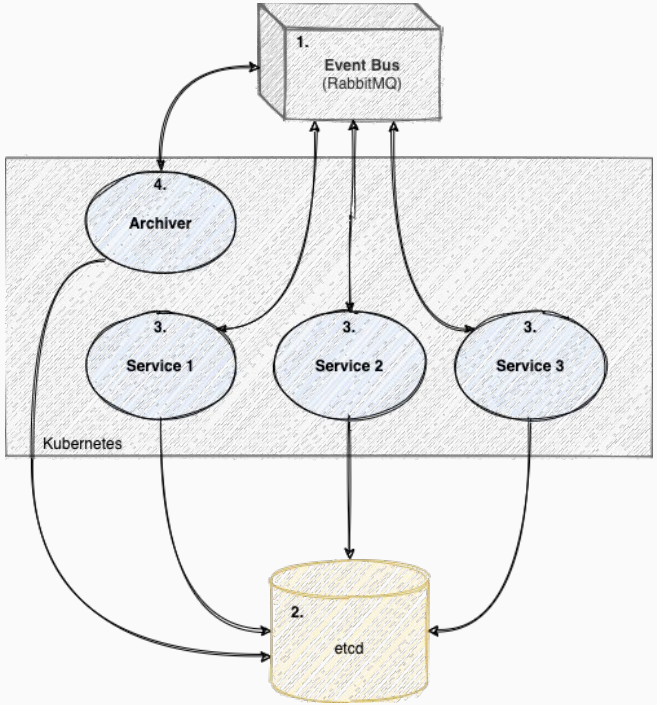




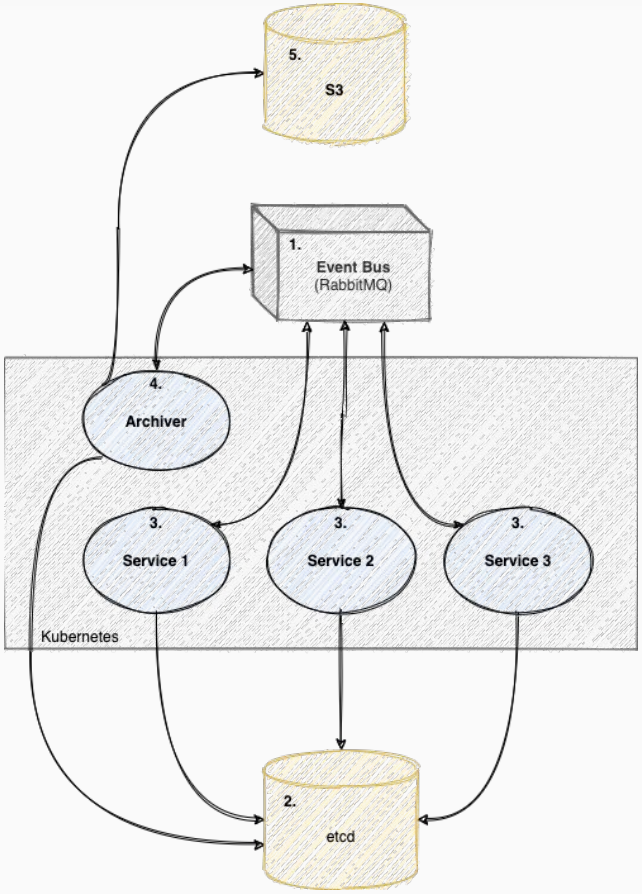
3. Event Driven From Scratch



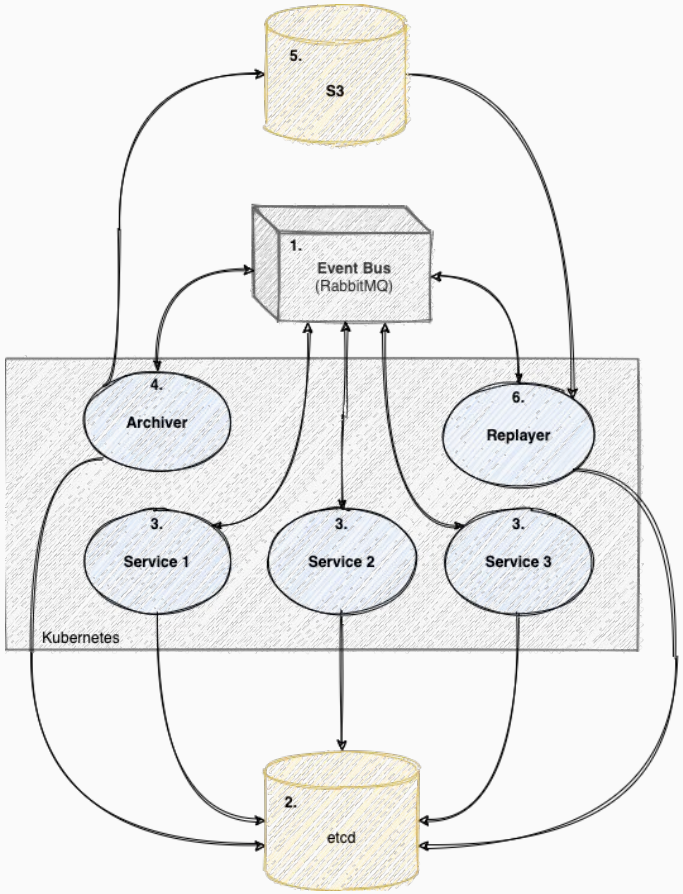
4. Event Driven From Scratch



5. Event Driven From Scratch



6. Event Driven From Scratch



Emit/Consume/React -- in other words

reliability nirvana

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message
3. Async: *Mail service* **consumes** `USER_SIGNUP` event and sends a welcome email to user

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message
3. Async: *Mail service* **consumes** `USER_SIGNUP` event and sends a welcome email to user
4. Async: *Mail service* **emits** `WELCOME_COMPLETE` message

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message
3. Async: *Mail service* **consumes** `USER_SIGNUP` event and sends a welcome email to user
4. Async: *Mail service* **emits** `WELCOME_COMPLETE` message
5. Async: *Billing service* also **consumes** `USER_SIGNUP` message and creates a Stripe subscription for the user

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message
3. Async: *Mail service* **consumes** `USER_SIGNUP` event and sends a welcome email to user
4. Async: *Mail service* **emits** `WELCOME_COMPLETE` message
5. Async: *Billing service* also **consumes** `USER_SIGNUP` message and creates a Stripe subscription for the user
6. Async: *Billing service* **emits** `BILLING_COMPLETE` message

In other words...

1. Sync: *Frontend app* talks to a backend API to create a user
2. Both: *Backend API* creates a DB entry for the user and **emits** a `USER_SIGNUP` message
3. Async: *Mail service* **consumes** `USER_SIGNUP` event and sends a welcome email to user
4. Async: *Mail service* **emits** `WELCOME_COMPLETE` message
5. Async: *Billing service* also **consumes** `USER_SIGNUP` message and creates a Stripe subscription for the user
6. Async: *Billing service* **emits** `BILLING_COMPLETE` message
7. Async: *Audit service* **consumes** `USER_SIGNUP`, `WELCOME_COMPLETE` and `BILLING_COMPLETE` events and archives them

How does this translate to improved reliability?

(Almost) Everything is async!

Implementation Reality – Organizational

reliability nirvana

Implementation Reality: Organizational

- Hard: (re-)Defining the architecture (2-4 weeks)
  - Flow diagrams
  - Documentation
  - Examples
- Hard: Becoming a thought-leader for your org (on-going)
- Hard: Convincing leadership (2-4 weeks)
- Medium: Convincing developers (1-4 weeks)
- Medium: Assisting developers (on-going)
- Hard: Communicating the architecture across engineering (4+ weeks)
  - Prepare talks, presentations

Batch Diagram

reliability nirvana

Batch.sh uses an event-driven architecture