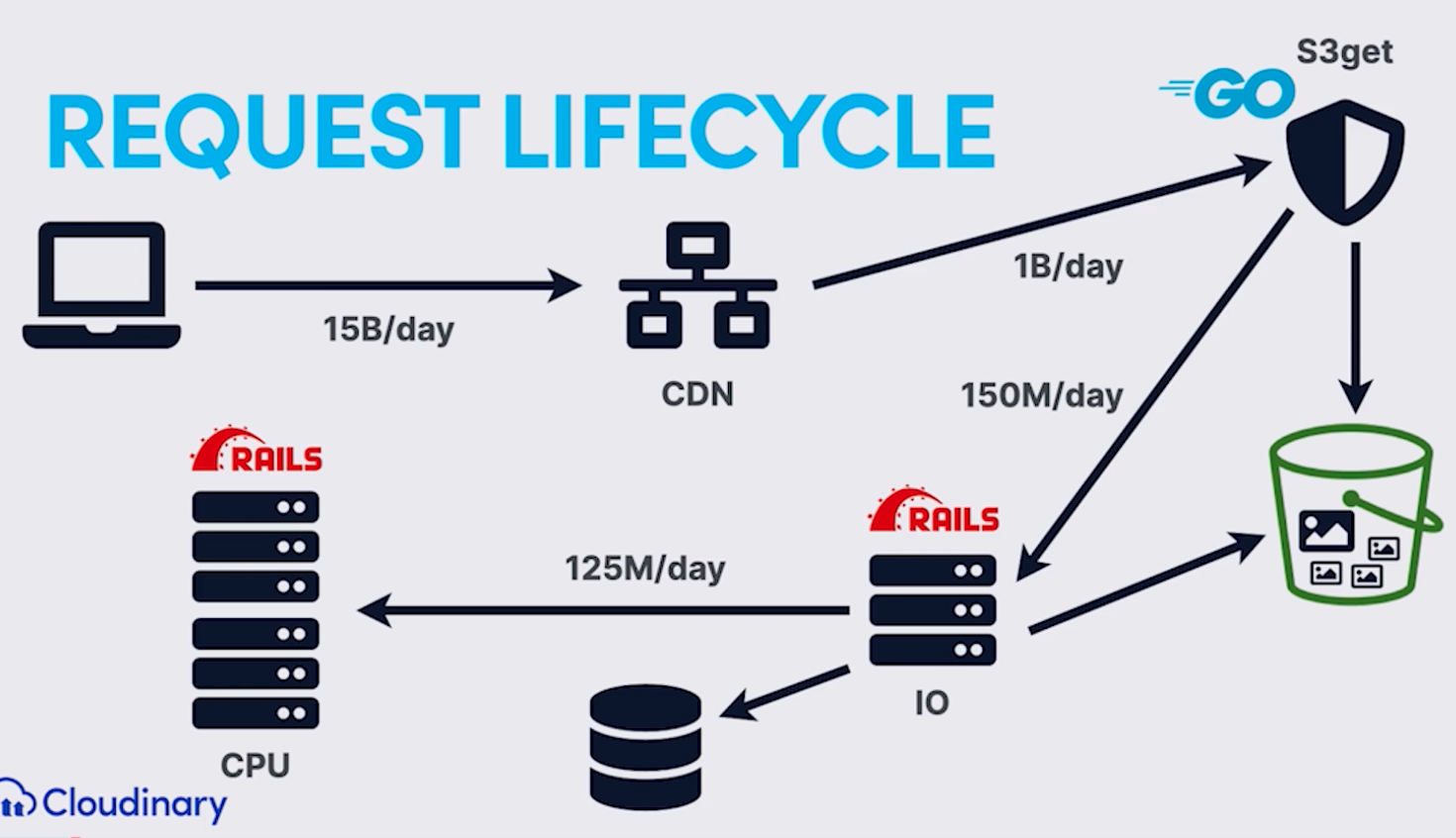
Methods for scaling:

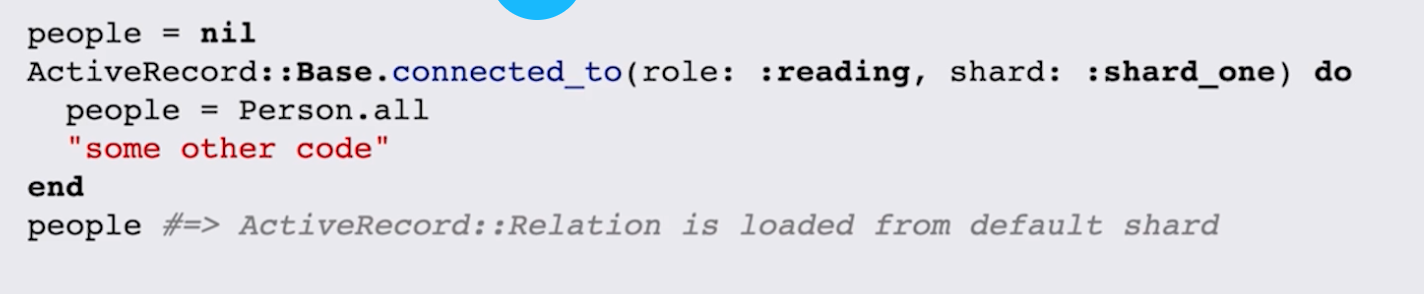
* Layers
* Sharding
* Location
* Deduplicating work
* Not scaling
* Human factors

Layers:

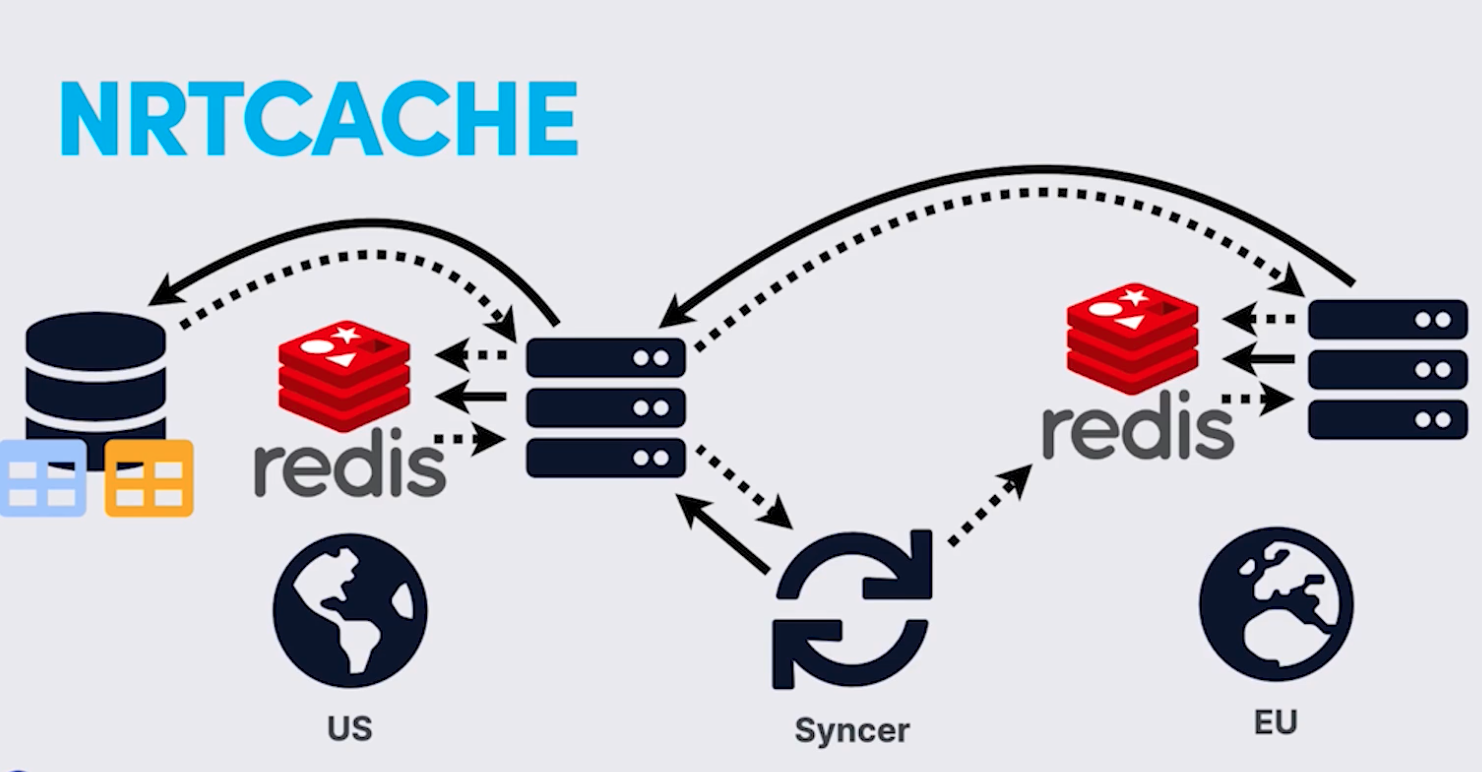
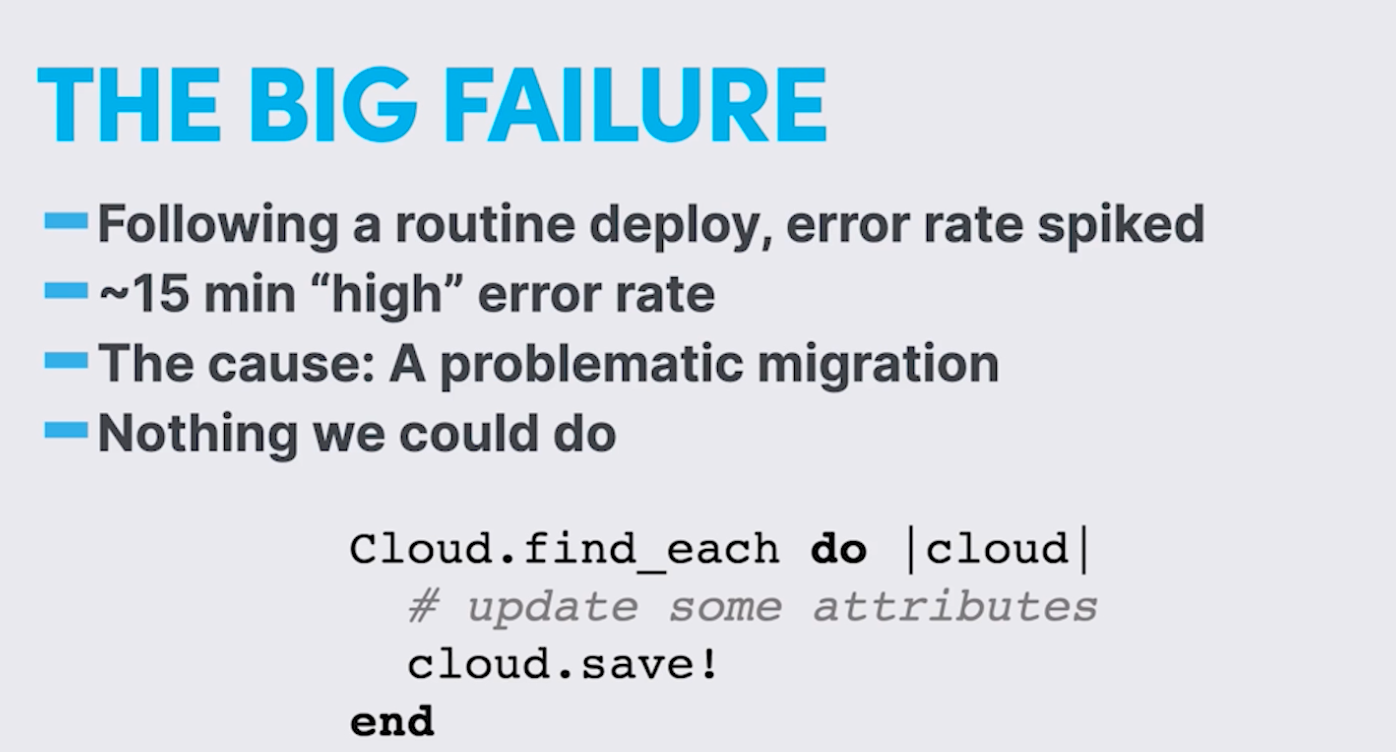


* 14/15 cases it will be stored in CDN
* CDN
  + Building your own vs buying
  + ~95% of traffic isn’t the problem
  + Pro: leverage best-in-class service/features
  + Pro: Multi-CDN for reliability (failover)
  + Con: Need to play by their rules
    - Write custom rules (f\_auto)
    - Every CDN provider has their own invalidation system for caching
    - Need to parse log files to bill customers
* S3Get
  + High-throughput, simple GO service
  + Handles 85% of requests it receives
  + Takes up to ~10% of computing resources vs IO
  + Pro: it’s fast
  + Con: Need to duplicate some Ruby logic in Go and vice versa
* Other layer advantages
  + Layers can scale independently
    - Horizontal slices also exist (image vs video)
  + Security (CPU can’t access internet or DB)
    - CPU layer can’t access other parts

Sharding

* What is sharding?
  + Vertical partitioning != sharding
    - Each database contains different tables
    - Increase simultaneous writes/reads
    - Supported in rails 6.0
  + Horizontal partitioning = sharding
    - 1 table distributed in multiple databases
    - Keeps table size under control
    - Supported in rails 6.1
* Sharding at Cloudinary
  + 1 main shard for app-wide data
  + Every cloud lives 100% on 1 of several shards
  + Code includes thousands of shard references
  + Test environments must be sharded too
* Pro: It’s fast
* Pro: Flexibility
* Con: Error-prone
  + 
* Need to execute code on the correct shard

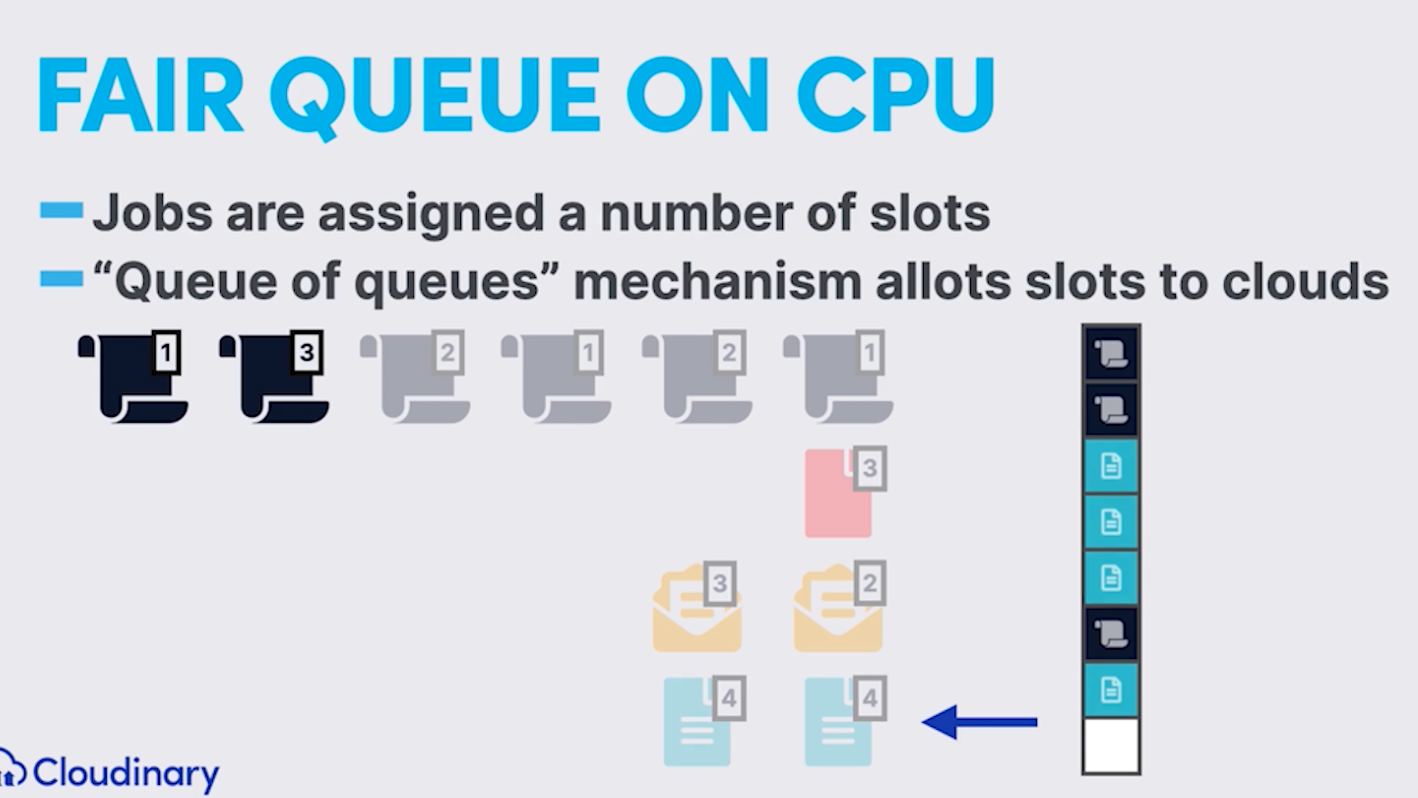
Location

* Regions at Cloudinary
  + 3 Regions: US, EU, AP
* Dedicated shards per region
* What about the primary DB?
  + Option 1. Run 3 completely independent systems
  + Option 2. EU and AP will be a little slower
  + Option 3. Multi-Primary DB
  + Option 4. NrtCache (Near real time)
* NrtCache
  + 
* 
  + Solution: Review every code change for cloud updates
  + Migrate to new and improved NrtCache
* Pro: Multi-region is great for customers/users
* Con: hard to do right safe and reliably

Deduplicating Work

* Locking
  + Goal: Don’t repeat transformation
  + Never block a job (2x > 0x)
  + Implementation: Best effort locking system
  + LOPTR (internal tool)
    - Read lock on asset before working on it
      * No write while lock is held
    - Write lock on derivation before generating
      * This process can write
      * Exclusive
    - Depends on a well-behaved client. Client must understand the lock process
    - In-memory lock table for speed
    - Written in scala for high concurrency
    - Failure to release
      * Timeout
    - Downtime
      * Pretend every lock request succeeded
      * Doesn’t work for data consistency but for availability
    - Scaling LOPTR
      * Cluster with request targeted by consistent hash
    - Pro: Resilient to traffic surges
    - Con: Unreleased locks can cause timeouts
    - Note: Not 100% reliable (but good enough)

Not Scaling

* Limit individual customer impact
  + Would you rather deal with:
    - 1 dissatisfied customer
    - Thousands of dissatisfied customers
* Rate limits through API
  + Quiet rate limits on clients
* Manage scarcity while eliminating scarcity
  + Fair queueing?
* Rate limits
  + Heavy API calls have strict limit
  + Locking effectively throttles non-rate-limited calls
* Fair Queue on CPU
  + Jobs are assigned a number of slots
  + “Queue of queues” mechanism allots slots to clouds
  + 
  + Prefer sync to async requests
* Fair queue for DB access
  + Queue of queues throttles per-cloud concurrency
  + One big monkeypatch on ActiveRecord
* Background jobs
  + Spread out impact for event
  + Anything that can be done out of band, should be
    - Examples:
      * CDN invalidations
      * Webhooks
      * Eager transformations

Human Factors:

* Scaling via humans
  + Education
    - Encourage practices like eager generation
  + Relationships
    - Understand customer use cases
    - They inform us about changes in use patterns
  + Look for win-wins!

Rails is challenging

* Upgrading Rails is hard
* Upgrading monkeypatched Rails is harder