facebook

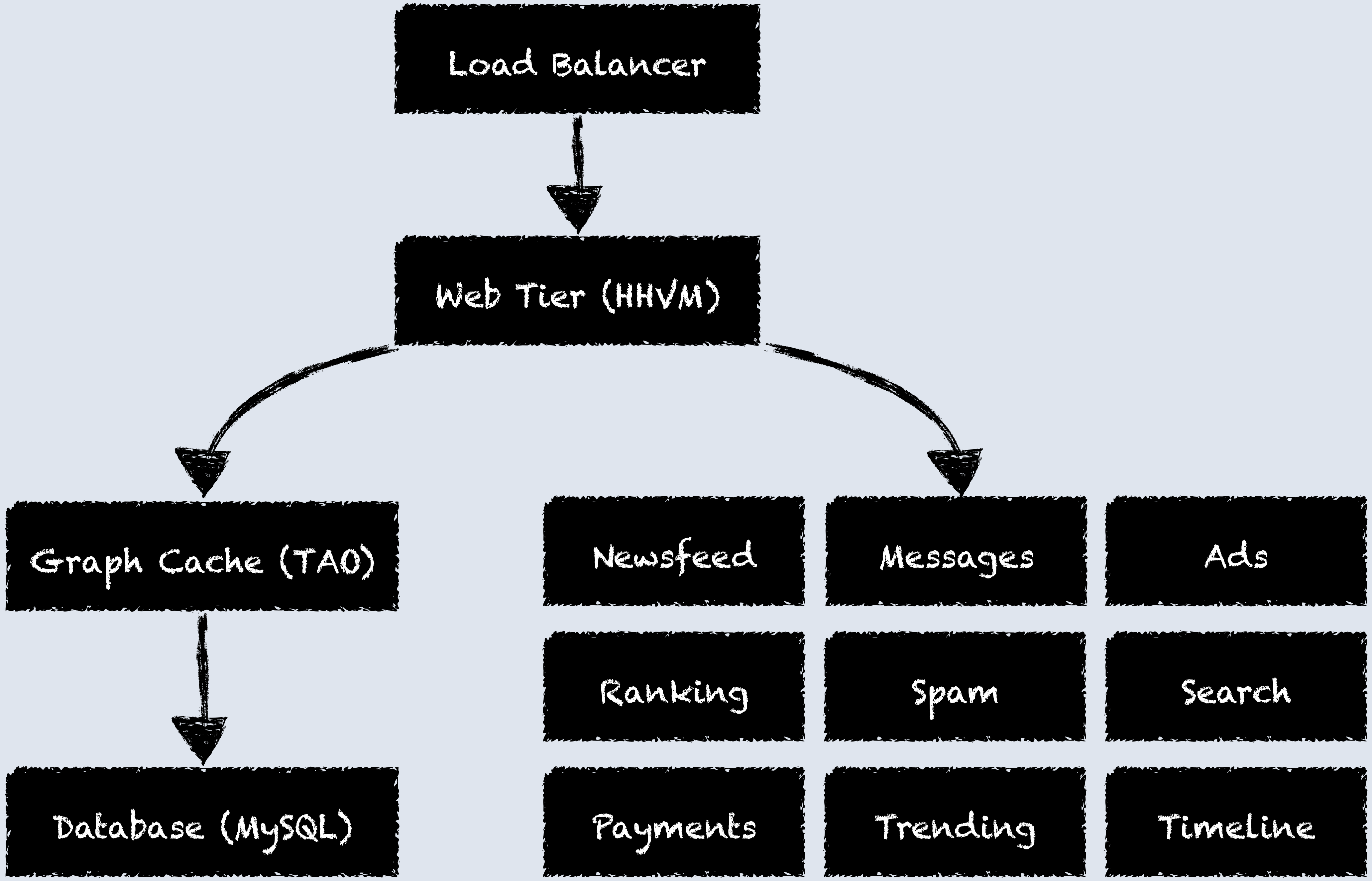
Systems @ Facebook Scale

Ben Maurer
bmaurer@fb.com

About Me

- At Facebook since 2010
- Co-founded reCAPTCHA
- Tech-lead of Web Foundation team
- Responsible for the overall performance & reliability of Facebook's user-facing products
  - Proactive — Design
  - Reactive — Outages

Facebook in 30 Seconds



Rapid Change

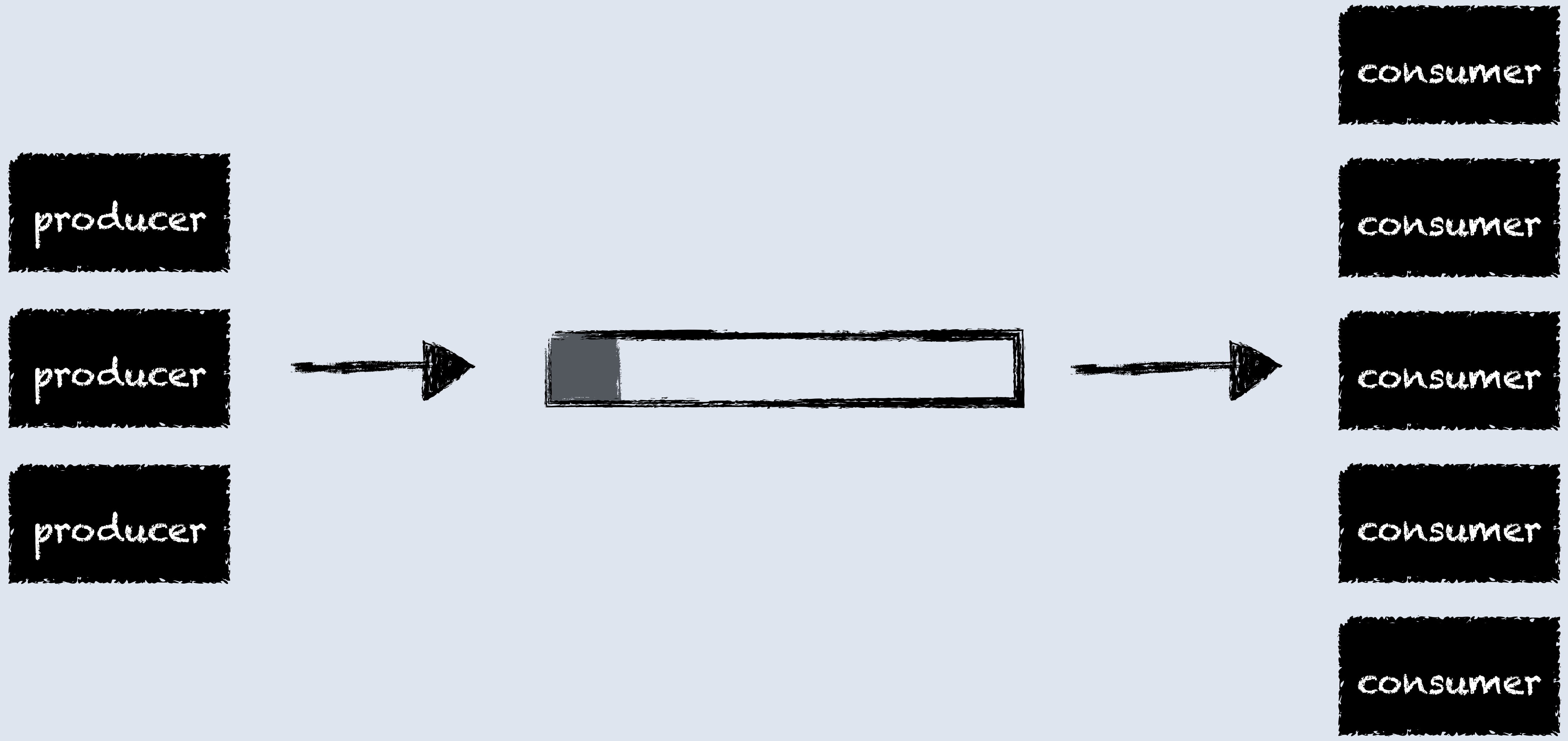
- Code released twice a day
- Rapid feature development — e.g. Lookback videos
  - 450 Gbps of egress
  - 720 million videos rendered (9 million / hour)
  - 11 PB of storage
  - Inception to Production: 25 days

Proactive Design for Scaling

- Solve scaling problems only once
- All Facebook projects in a single source control repo — easy code reuse
  - Folly: base C++ library
  - Thrift: RPC $\longrightarrow$ All open source!
  - Proxygen: HTTP(S) server
  - RocksDB: persistent key-value store

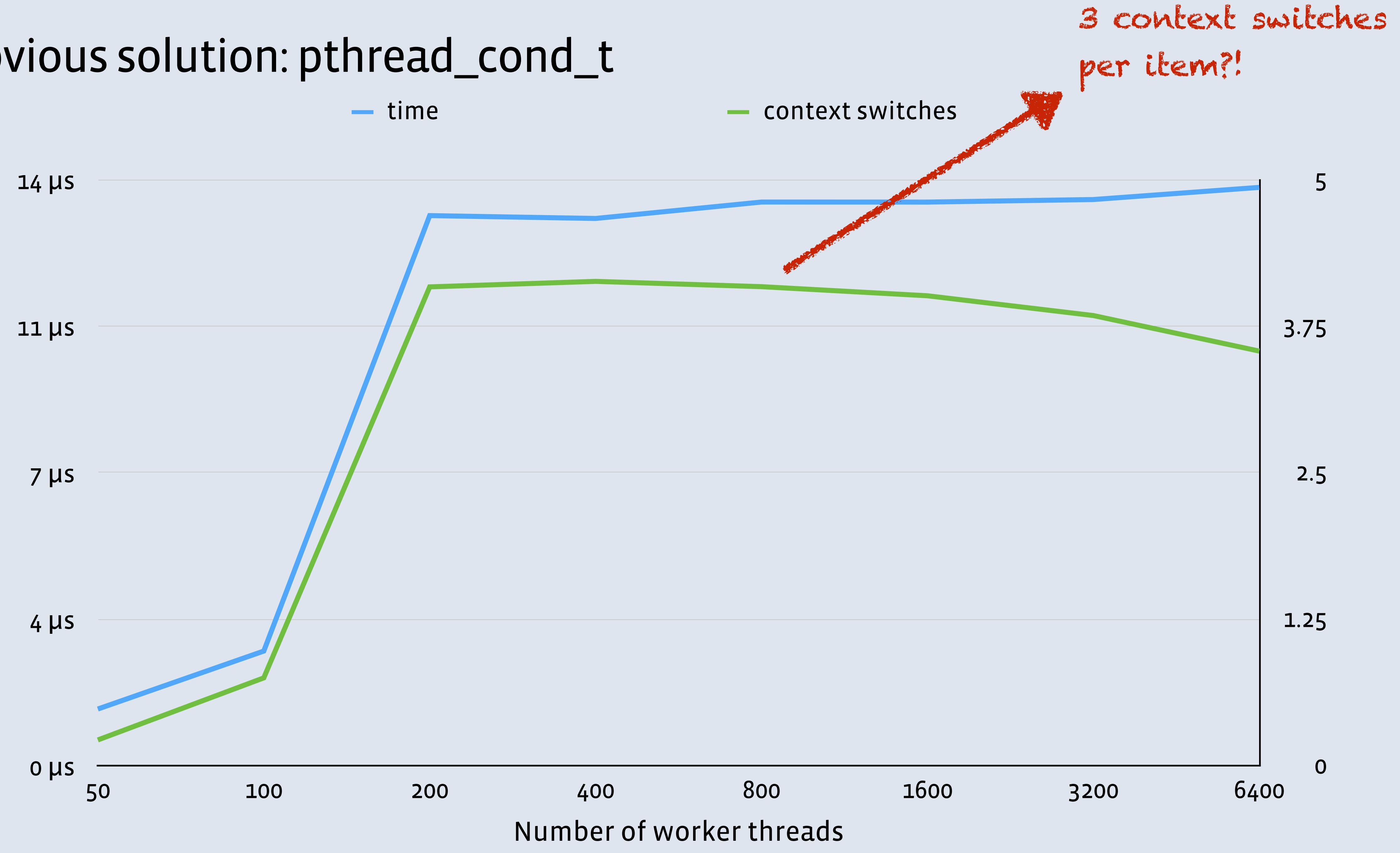
Efficient Synchronization

Producer/Consumer Queues



How to Implement Producer/Consumer

- Obvious solution: pthread_cond_t



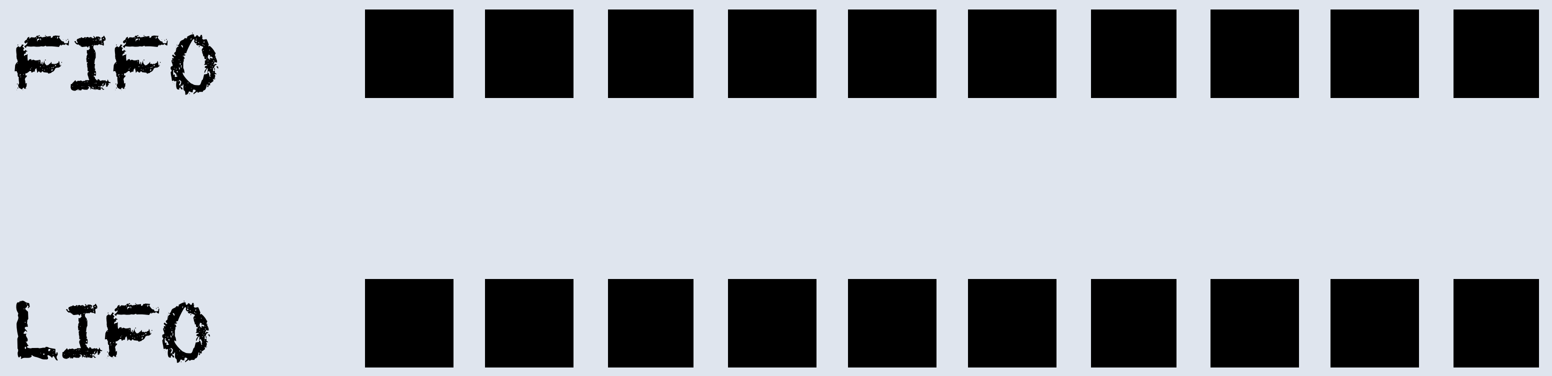
Multiple Wakeups / Deque

Potential context switches

```
pthread_cond_signal() {
  lock();
  ++futex;
  futex_wake(&futex, 1);
  unlock();
}

pthread_cond_wait() {
  do {
    int futex_val = cond->futex
    unlock();
    futex_wait (&futex, futex_val);
    lock();
  } while (!my_turn_to_wake_up())
}
```

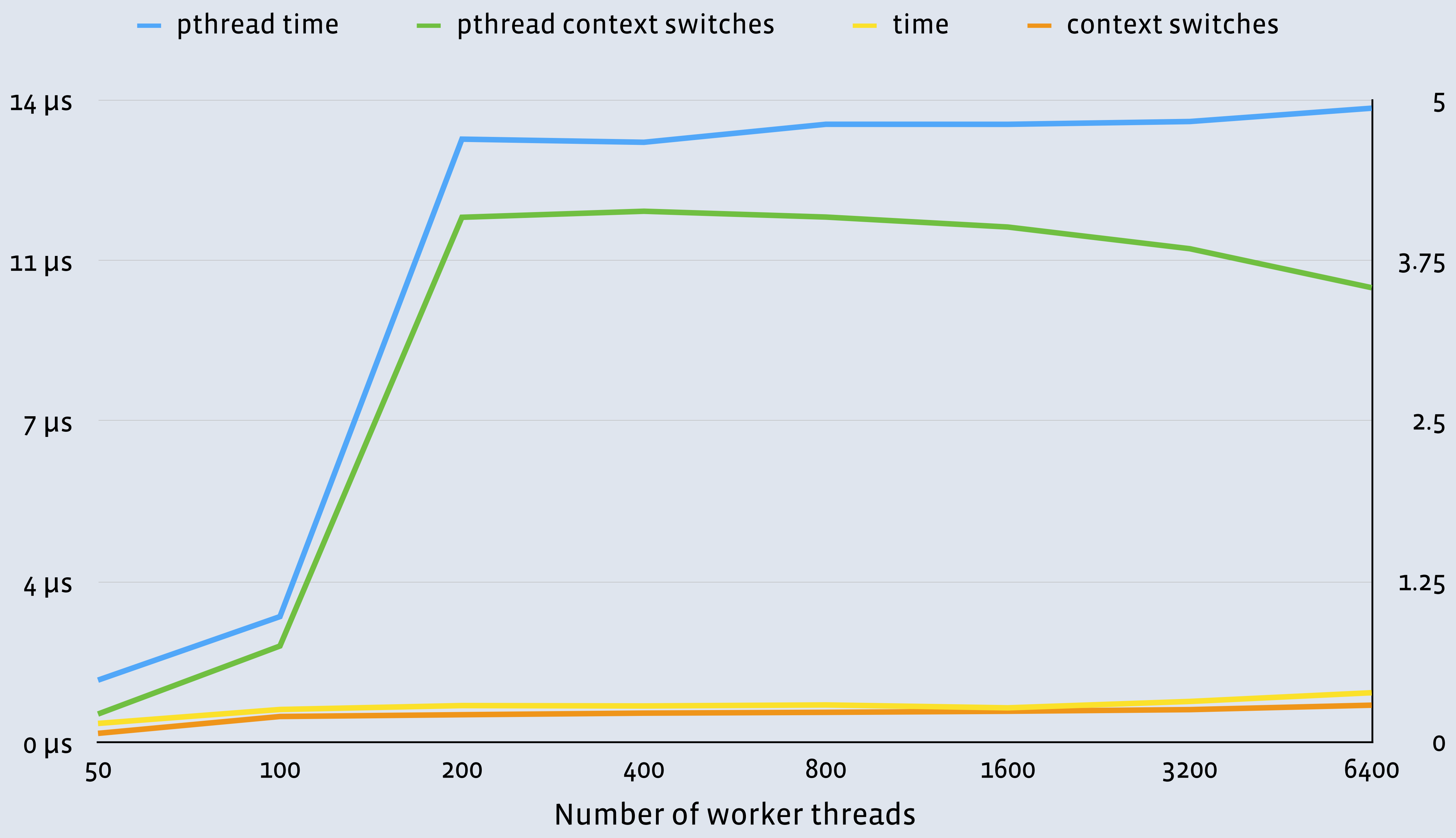
Worker Thread Ordering



- FIFO: New work is schedule on the thread that has been idle longest
  - Bad for memory usage
  - Bad for the CPU cache
- 3-5% CPU efficiency wins switching from FIFO to LIFO
- Save memory by releasing resources from idle threads

LifoSem

- 13x faster. 12x fewer context switches



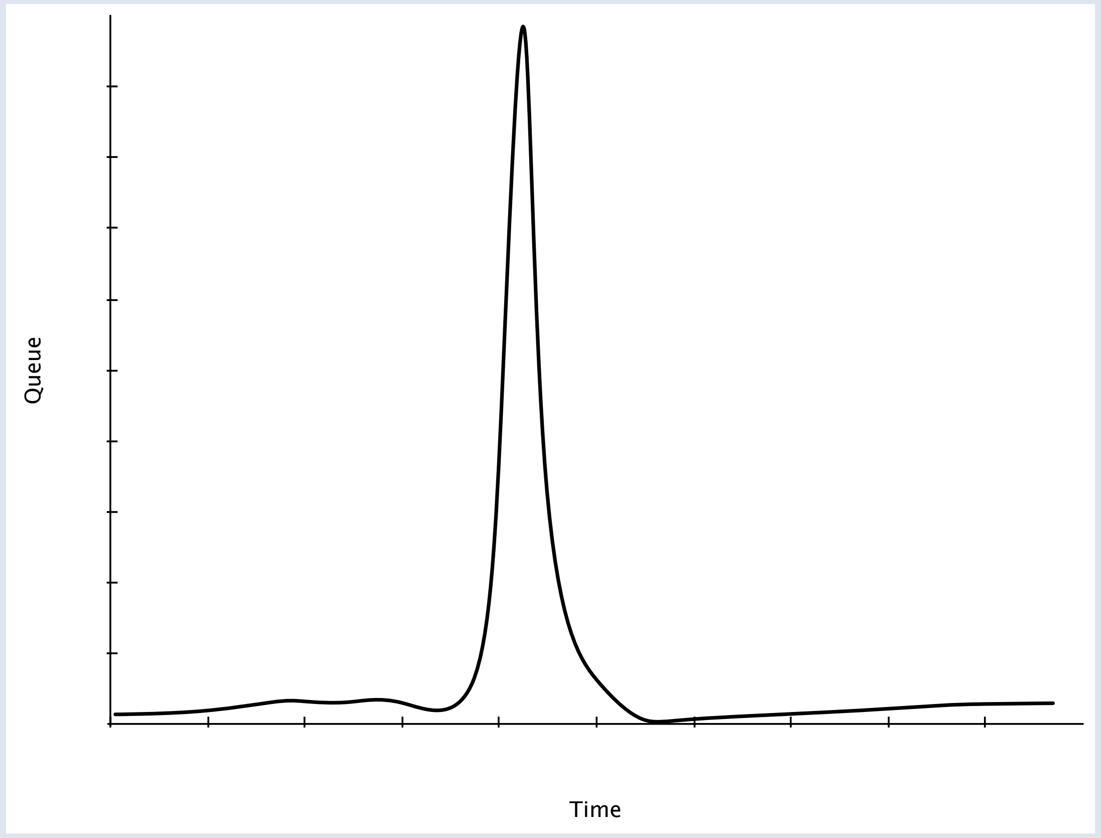
Synchronization Performance

- pthread_cond_t not the only slow synchronization method
- pthread_mutex_t: can cause contention in futex spinlock. A different futex api could fix this
- pthread_rwlock_t: Uses a mutex. Consider RWSpinLock in folly

Managing Queues

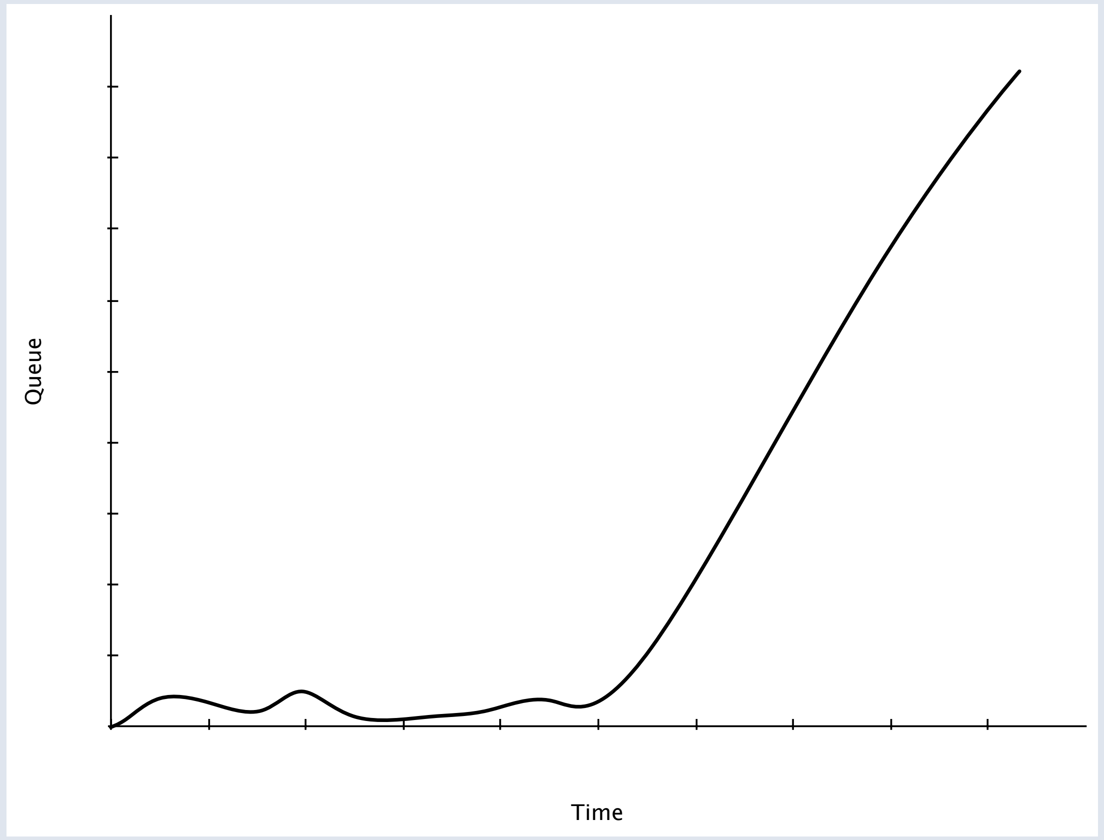
Good queueing

- Deal with a burst of load
- Processing speed > arrival speed
- Increase reliability



Bad queueing

- Server is overloaded
- Processing speed < arrival speed
- Causes latency



Overload Handling Philosophy

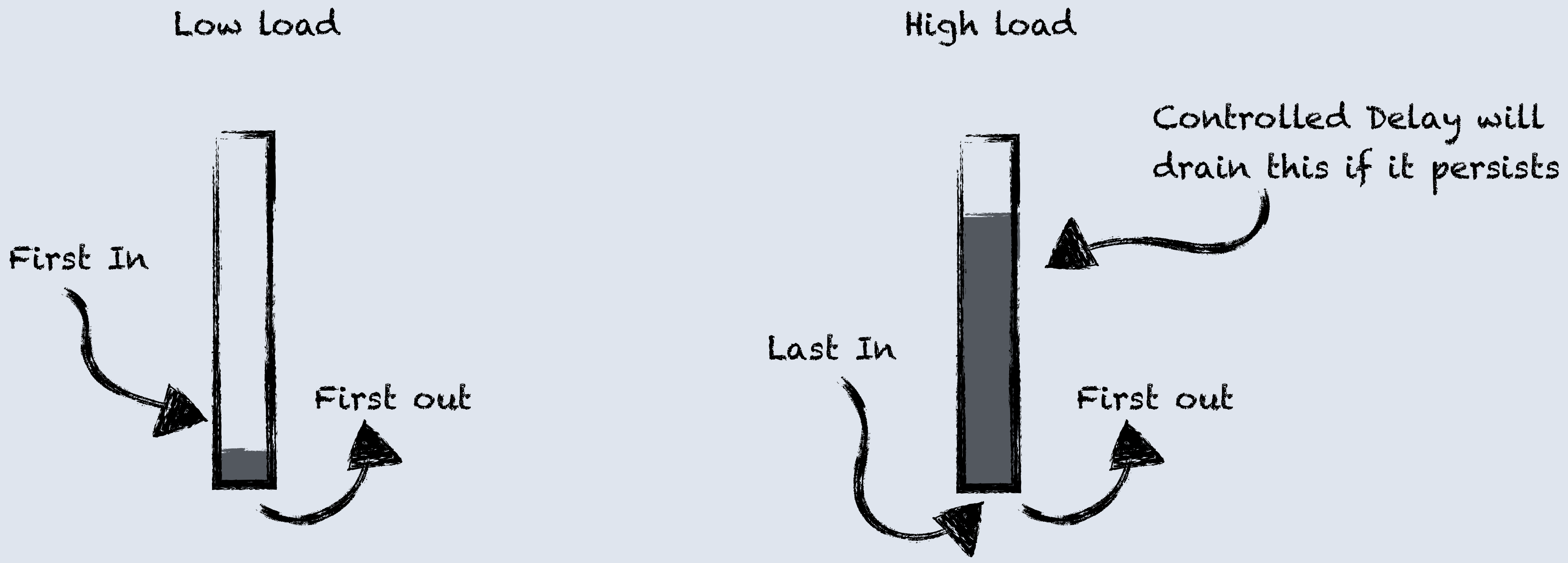
- **If you’re going to fail, fail quickly** — prevents delaying the overall request
- **Servers should signal overload** — OK to say “I can’t help you right now”
- **Order doesn’t matter** — servers need not first-come-first-serve
- **Clients should defend themselves** — don’t rely on the server
- **Complex knobs are tuned poorly** — design parameter-free abstractions

Controlled Delay

- Inspired by CoDel algorithm to reduce buffer-bloat
- **Key insight:** processing speed > arrival speed implies queues frequently become empty
- **Solution**: Controlled delay logic
  - Check if the queue has been drained within the last N seconds
  - If not set queue timeout to 10-30ms

Adaptive LIFO

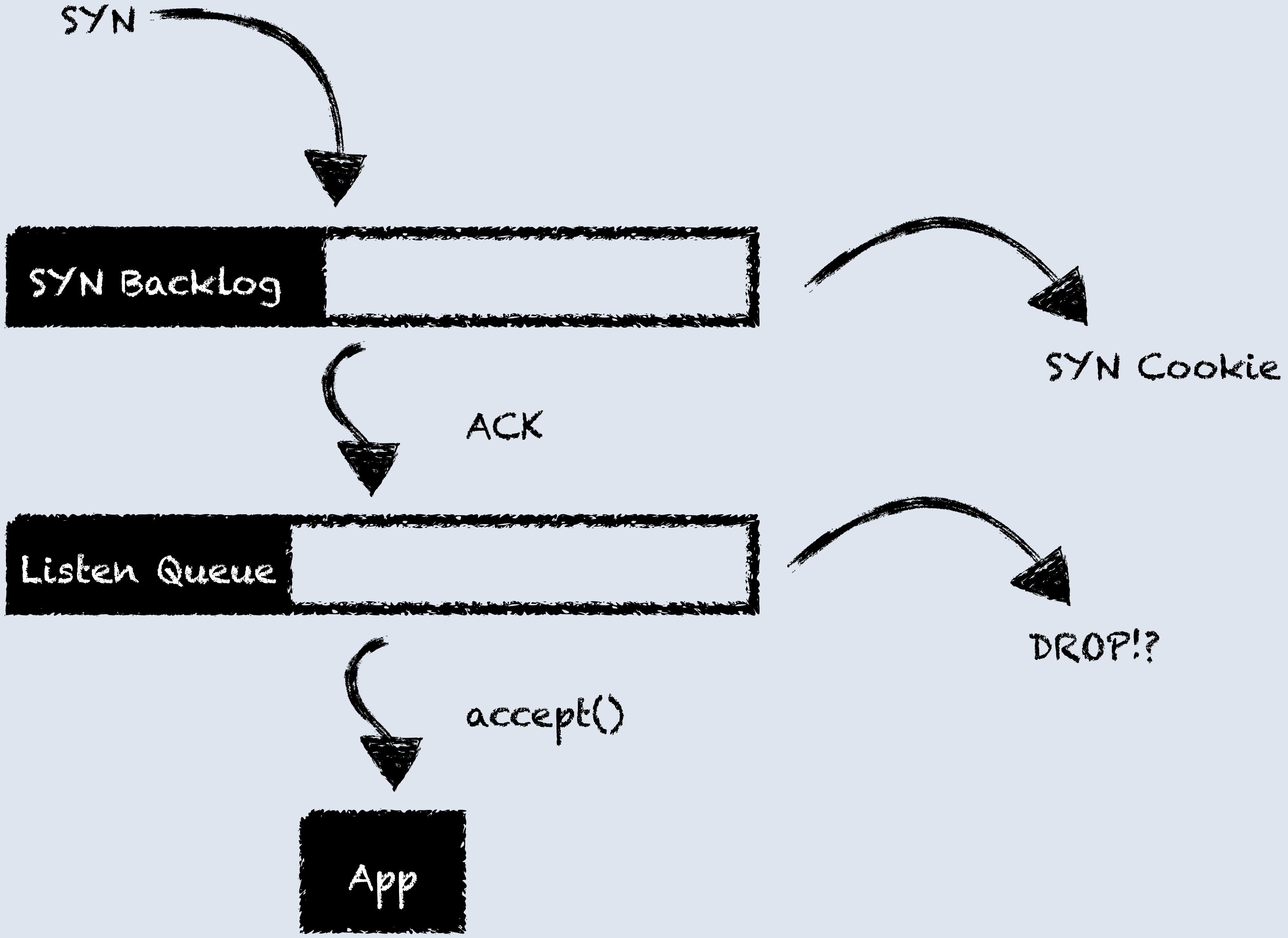
- FIFO: “first in” request has already experienced the most latency — user may have abandon request.
- **Solution**: switch to LIFO mode if large number of items in the queue



Concurrency Control

- Teach clients to protect themselves by recognizing when the services they talk to are unresponsive
- Approach: measure & cap the number of active requests to a given service
- Protects the throughput of a service that calls a heterogeneous set of other services

Listen Queue Overflow



SYN Queuing: A Setting You Can Fix Today

```
sysctl -w net.ipv4.tcp_abort_on_overflow=1
```

- When Linux gets a SYN, if there are too many connections waiting to be accepted it will simply drop the packet
  - **Theory**: the client will see the packet was dropped, eventually retransmit
  - **Practice**: the server is overloaded. Trying again won't help. Besides, it will usually take 1 second for the server to try again.
- We open sourced a patch to do this better
- **This setting has stopped Facebook from going down**

Responding to an Incident

Keeping the big picture in mind

Focus on communicating and brainstorming

Questions to ask

- What is everybody doing right now? What's our plan of attack?
- What is the current hypothesis? What would prove/disprove that?
- How can we quickly mitigate the impact? Can we revert?
- Is there an alternate strategy we can pursue?
- What will we need to do in 10 minutes? Can somebody start that?
- Who else should we call?

Learning from an Incident

- Incidents are a great chance to improve the reliability of your system
- Facebook does a weekly incident review. Many teams do their own internal reviews as well
- **Goals**
  - Share knowledge of what happened
  - Document how the incident was handled
  - Come up with action items to make Facebook more reliable

How not to review an incident

- Make people feel bad about being “invited”
  - Everybody makes mistakes. Incident review is not about determining who is doing a bad job. It is to discover ways to improve
- “Next time we’re not going to do that”
  - Well, obviously
  - Assume you’ll break things a different way next time. What can you fix generically?

DERP!

- **Detection**
  Did you detect the incident automatically? Did your alarms and dashboards let you quickly detect the issue?
- **Escalation**
  Did the right people get involved quickly? If you had to call a different team to help with the issue, could they have been alerted automatically?
- **Remediation**
  What did you need to do to fix the issue? Can these steps be automated?
- **Prevention**
  What improvements could remove the risk of this type of failure happening again? How could you have failed gracefully or failed fast to reduce the impact of this failure.

Think bigger than yourself

- Common to think about action items that affect your team
- Important to push global changes that make things safer for everybody
- Example: rarely used but critical feature fails, not noticed in noise
  - Intuition: “we should really monitor failure for feature X”
  - Think bigger: “we should adjust our monitoring system to detect the percentage failure in more rarely used features”
- Experienced incident members should surface these types of action items

How We Handle Facebook Scale

Proactive
Create battle-tested abstractions that solve known scalability challenges.
Make the easiest way to build services the most scaleable way.

Reactive
Avoid getting stuck during an incident by keeping the big picture in mind.
Review incidents and ensure lessons learned used in the proactive mission.

facebook