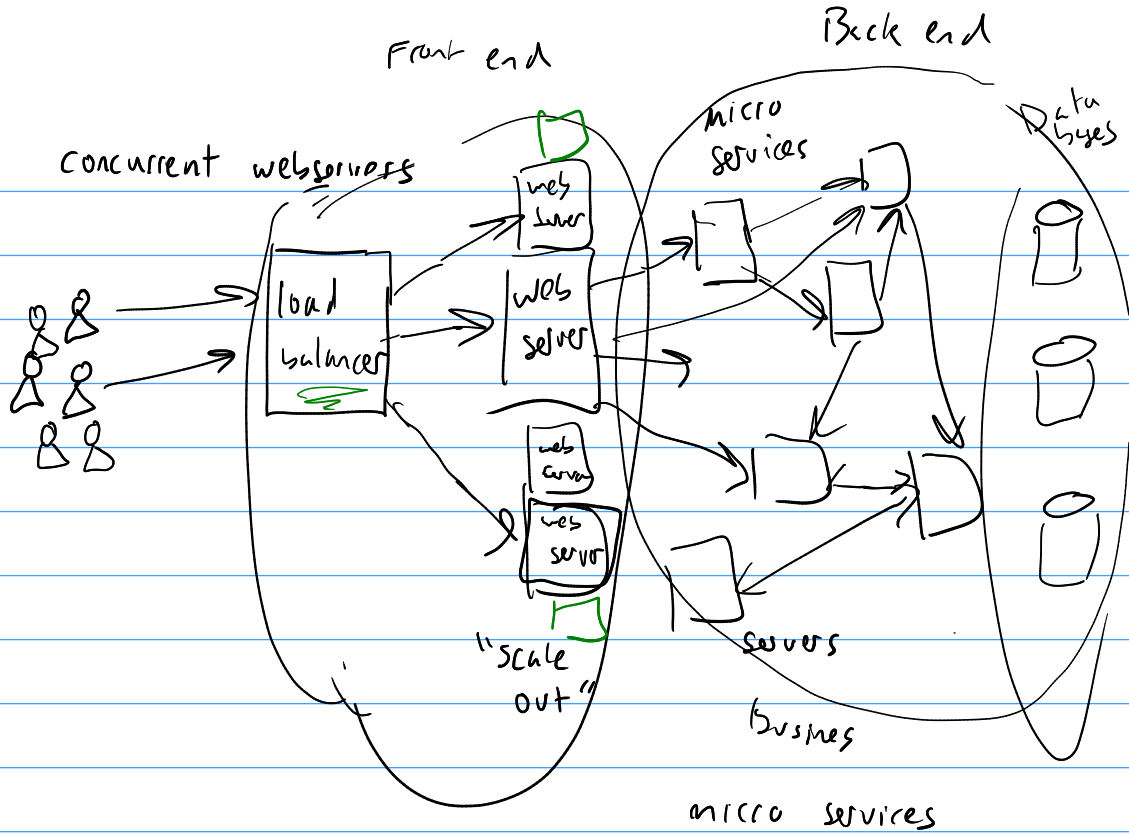
CS 3214 concurrent webservers

SPOT



3-tier

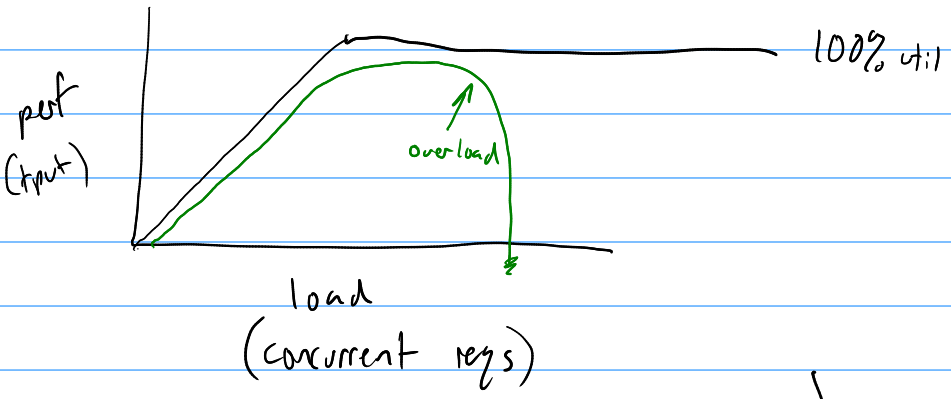
The slow way

- "iterative server": one client req at a time
  - wait even if the server blocks
  - single CPU
  - clients see high latency
  - server has low util

```
for {
    get req
    do it(req)
}
```

Concurrency { vs. threads / events

1978 equivalent but tradeoffs in practice



Thread-based

- each client request own thread
  - → underlying system (context, state)
    - e.g. BLOCKED

PROs
+ familiar control flow (blocking fine)
+ UNIX fds make things easy
+ processes can give us isolation

Apache

CONS
- concurrency-related bugs are hard
- contention on resources
- tuning concurrency (how many threads)

Events-based
→ Never block! 100% utilization

NGINX

reorganize the program
allow app to decide what to do when

how to avoid blocking?
how to manage the tasks?

1. identify blocking points
   split the code there, but keep track of "state machine"
2. event loop
   figure out when events happen → new request
   new ~~response~~ data
   connection closed

HTTP/1.0

   → select(fd is it "ready"
   poll
   epoll ← Linux

reads ↓ writes sometimes block
(TCP flow control)

Non-blocking fds

open(... O_NONBLOCK)
→ EWOULDBLOCK

stack ripping

libraries: libevent, libuv ↗ node.js

provide an abstraction on top of event loop

High-level lang support async/await in JS

multithreading ← great for programming

event-based outperform (NGINX vs. Apache) cost of programming complexity