CS 5600 · Spring 2026 · Team 0100

StreetLegal Webserver

Traffic is never smooth.

Jose Diaz · Malik Vernon · Seretsi Lekena · Srashti Gupta

LANGUAGE

Rust

stdlib only — no crates

ARCHITECTURE

Thread Pool

fixed + adaptive

QUESTION

P99 ≥ 30%?

burst latency reduction

The Fixed Pool Dilemma

BURST TRAFFIC

Queue Floods

100 requests arrive. 8 workers. 92 wait in line. P99 explodes.

workers: 8 | queue: overflowing

QUIET PERIOD

Threads Idle

2 requests arrive. 8 workers spin-wait. CPU wasted on nothing.

workers: 8 | queue: empty

You pick a number before the server starts and pray it's right.

Pick too small → latency spikes. **Pick too big → memory wasted.**

What Existing Systems Do…

APACHE HTTPD

Pre-fork Processes

OS manages pool sizing via MinSpareServers / MaxSpareServers. Heavy per-process overhead.

NODE.JS

Event Loop

Single thread + async I/O. Never blocks. CPU-bound work starves the loop.

GO NET/HTTP

Per-request Goroutine

Spawn a goroutine per request. Runtime scheduler handles concurrency. ~2 KB stack overhead each.

All dodge the problem differently. We ask: **can a hand-rolled OS-primitive pool do better for burst workloads?**

HYPOTHESIS

The Research Question…

Can a thread pool that dynamically sizes itself based on queue depth and latency reduce P99 tail latency under bursty traffic — using only Rust stdlib?

≥ 30%

predicted P99 reduction

3 trials

burst · scale · latency

Producer-Consumer at the Core

PRODUCER BLOCKS WHEN…

Queue is full

`not_full.wait()`

CONSUMER BLOCKS WHEN…

Queue is empty

`not_empty.wait()`

SHUTDOWN SIGNAL

Both condvars get

`notify_all()`

DESIGN DECISION

Why Mutex + Condvar, not Channels

❌ STD::SYNC::MPSC CHANNELS

- Unbounded → no back-pressure
- Bounded → single condition, can't separate full vs. empty
- Multi-consumer needs Arc<Mutex<Receiver>> anyway
- Can't inspect queue depth without wrapping

✅ MUTEX<VECDEQUE> + 2 CONDVARS

- `not_full` wakes producers precisely
- `not_empty` wakes workers precisely
- `closed` flag for graceful drain
- `len()` readable under the same lock

Classic OS monitor pattern — the same primitive taught in OSTEP, applied directly.

Scaling Signals — the Two Dials

DIAL 1 — QUEUE DEPTH

Back-Pressure Signal

≥ 5% full	+50% workers
≥ 15% full	2× workers
≥ 30% full	4× workers

DIAL 2 — AVG LATENCY (MS)

Saturation Signal

≥ 5 ms	noticeable slowdown
≥ 20 ms	degraded
≥ 50 ms	saturated → 4×

`all_busy` override: if every worker is active and *any* job is queued → scale up regardless of thresholds

No Crates, Just Std

Using out-of-the-box Rust. No async runtime, no concurrency helpers.

`std::net`

TcpListener

TcpStream

`std::sync`

Mutex · Condvar

Arc · AtomicUsize

`std::thread`

spawn · JoinHandle

sleep · yield_now

No Tokio. No Rayon. No crossbeam.

Every synchronisation primitive written by hand.

BoundedQueue — the Critical Structure

```rust
pub struct BoundedQueue<T> {
    inner:     Mutex<QueueInner<T>>,
    not_empty: Condvar,  // wakes workers
    not_full:  Condvar,  // wakes producers
    capacity:  usize,
}

// push blocks when full:
while inner.data.len() >= self.capacity {
    inner = self.not_full.wait(inner)?;
}
inner.data.push_back(item);
self.not_empty.notify_one();
```

NOT_EMPTY CONDVAR

Workers wait here when queue is empty. Producer calls notify_one() after every push.

NOT_FULL CONDVAR

Accept loop waits here when queue is at capacity. Worker calls notify_one() after every pop.

CLOSED FLAG

Set during shutdown. Both condvars get notify_all() so everyone wakes and exits cleanly.

The Worker Loop

```rust
pub fn run(id: usize, queue: Arc<..>, metrics: ..) {
    while let Some(job) = queue.pop() {
        // record queue wait time
        let wait_ms = job.queue_wait_ms();
        metrics.request_start();

        let start = Instant::now();
        handler::handle_connection(job.stream);

        metrics.request_end(start);
    }
    // queue returned None — shutdown signal
}
```

WHILE LET SOME(JOB)

Blocks on pop() when idle. Returns None when queue is closed — clean exit with no extra signalling.

METRICS RECORDED

Queue wait time · handle time · total time. Feeds into recent_avg_latency_us() for adaptive scaling.

DYNAMIC VARIANT

Checks retire_flag.load() after each request. Exits loop early when pool scales down.

Graceful Shutdown


```rust
fn shutdown(&mut self) {
    self.queue.close();                 // 1. signal
    for handle in self.workers.drain(..) {
        let _ = handle.join();          // 2. wait for clean exit
    }
}
```

No kill(). No unsafe. Workers finish their current request, then exit. In-flight connections are never dropped mid-response.

The Adaptive Extension

SCALE-UP (IMMEDIATE)

New threads spawned right away when compute_target() says pool is short. No warmup delay.

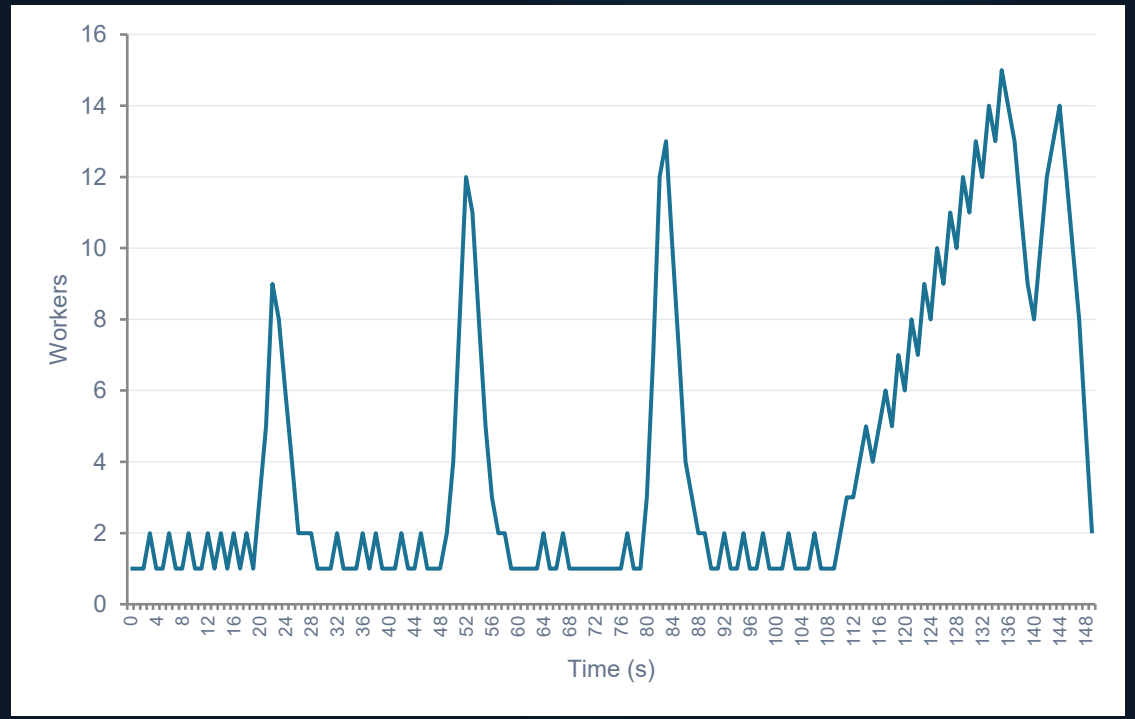
SCALE-DOWN (LAZY RETIRE)

Excess workers get retire_flag = true. They finish their current request then break. No blocking join() on the hot path.

DEAD HANDLE CLEANUP

scale_pool() calls is_finished() on retired handles and reaps them without blocking the accept loop.

WORKERS OVER TIME (DYNAMIC POOL)



Benchmarking with Apache Bench

```
# Scale test: 1000 requests across concurrency levels
ab -n 1000 -c 1   http://127.0.0.1:7878/index.html
ab -n 1000 -c 4   http://127.0.0.1:7878/index.html
ab -n 1000 -c 16  http://127.0.0.1:7878/index.html
ab -n 1000 -c 128 http://127.0.0.1:7878/index.html

# Burst test: baseline quiet → c100 spike → cool down (×3 cycles)
ab -n 5000 -c 2    ...   # 25s baseline
ab -n 1500 -c 100  ...   # 5s burst
```

LATENCY TEST

Low concurrency (c2–c32), 1000 req each. Measure P50/P90/P99 with no queue pressure.

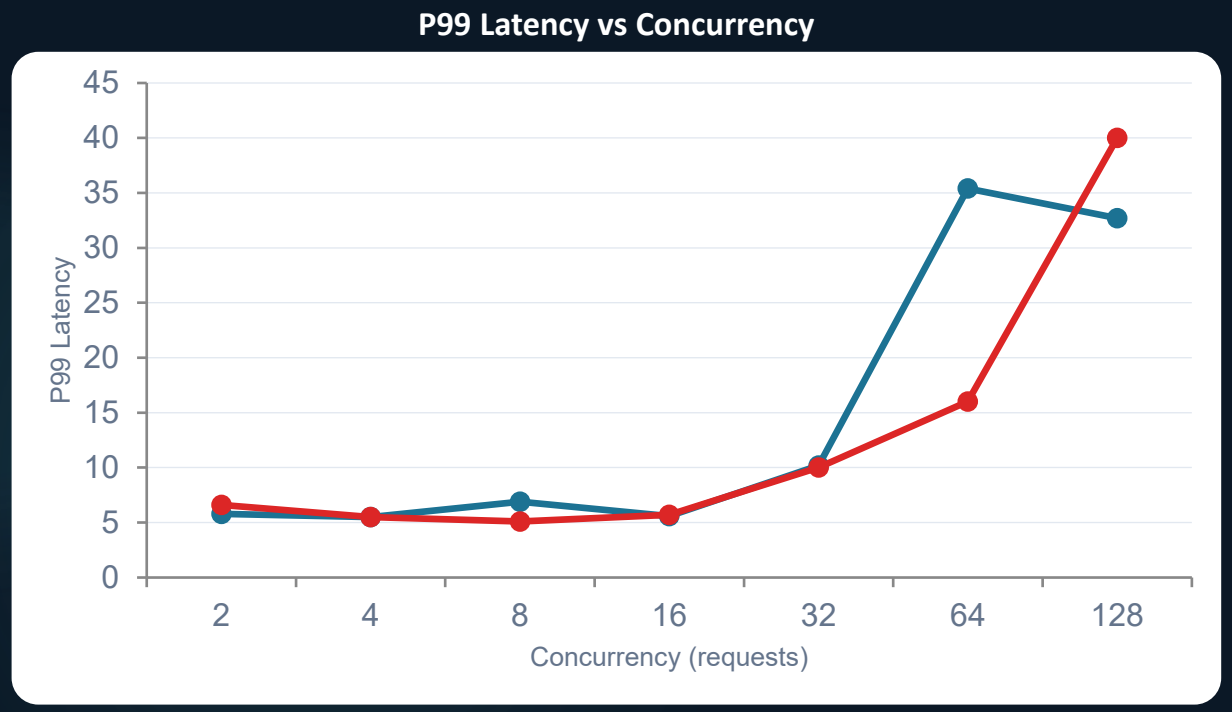
SCALE TEST

c1 → c128. Find throughput peak and where P99 starts diverging.

BURST TEST

3 spike cycles. The primary hypothesis test — fixed vs adaptive burst P99.

Three Latency Regimes



ZONE 1 — C1 TO C4

P99 < 7 ms

Queue rarely fills. Workers idle between requests. Latency is pure network + parse overhead.

ZONE 2 — C8 TO C32

P99 6–10 ms

Queue starts filling at burst moments. Workers are mostly busy. Occasional scheduling jitter appears.

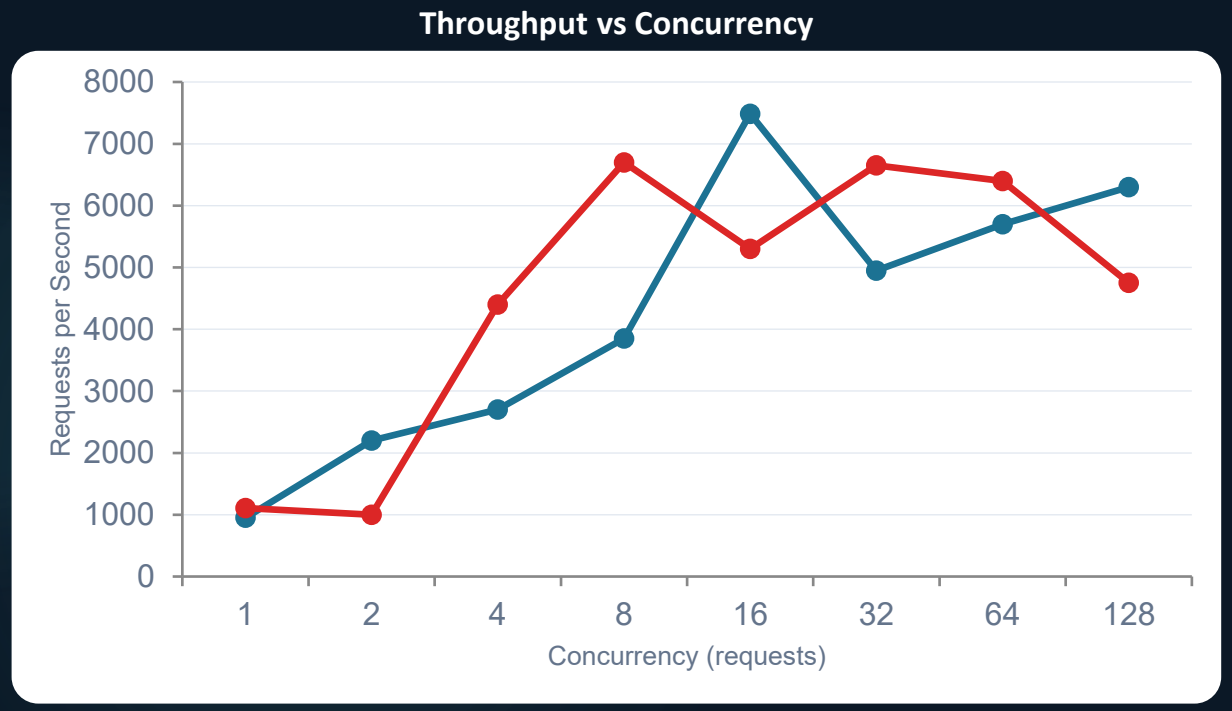
ZONE 3 — C64 TO C128

P99 jumps to 35 ms

Queue saturated. Requests pile up behind 8 workers. OS scheduler noise dominates the tail.

Fixed pool, averaged across 3 trials — 1000 req per concurrency level

Scale Test Baseline Numbers



CONCURRENCY	REQ/SEC	P50 MS	P99 MS
1	1,106	< 1	6.0
4	2,701	< 1	5.3
8	3,835	0.3	6.7
16	**7,487**	2.0	5.7
32	4,952	5.3	10.0
64	5,690	10.0	35.4
128	6,290	19.3	32.7

Peak: **7,487 req/s at c16** — matches the 8-worker pool saturation point

Fixed pool · averaged across 3 trials · 1000 requests per level

RESULTS

P50 Stable, P99 Noisy

P50 (MEDIAN)

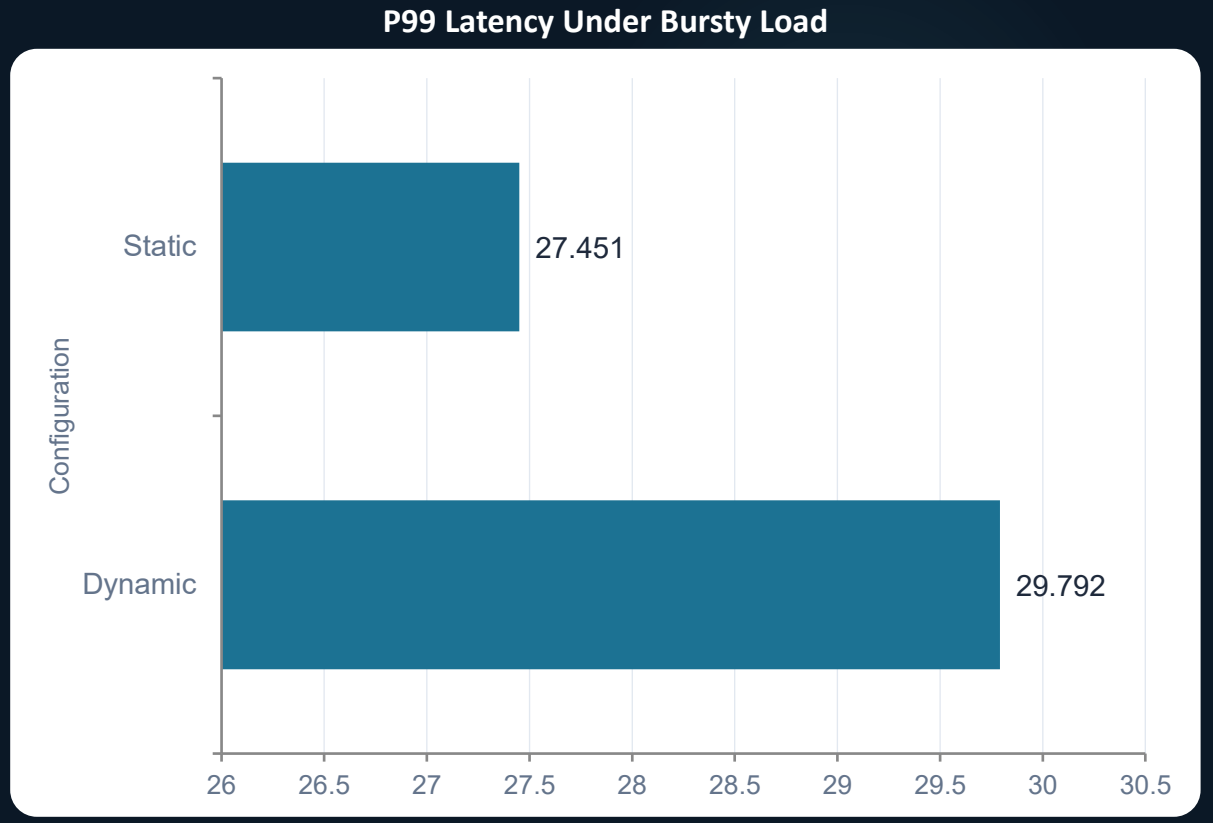
Stable

Stays < 20 ms even at c128. The average case is fine — workers keep up.

P99 (TAIL)

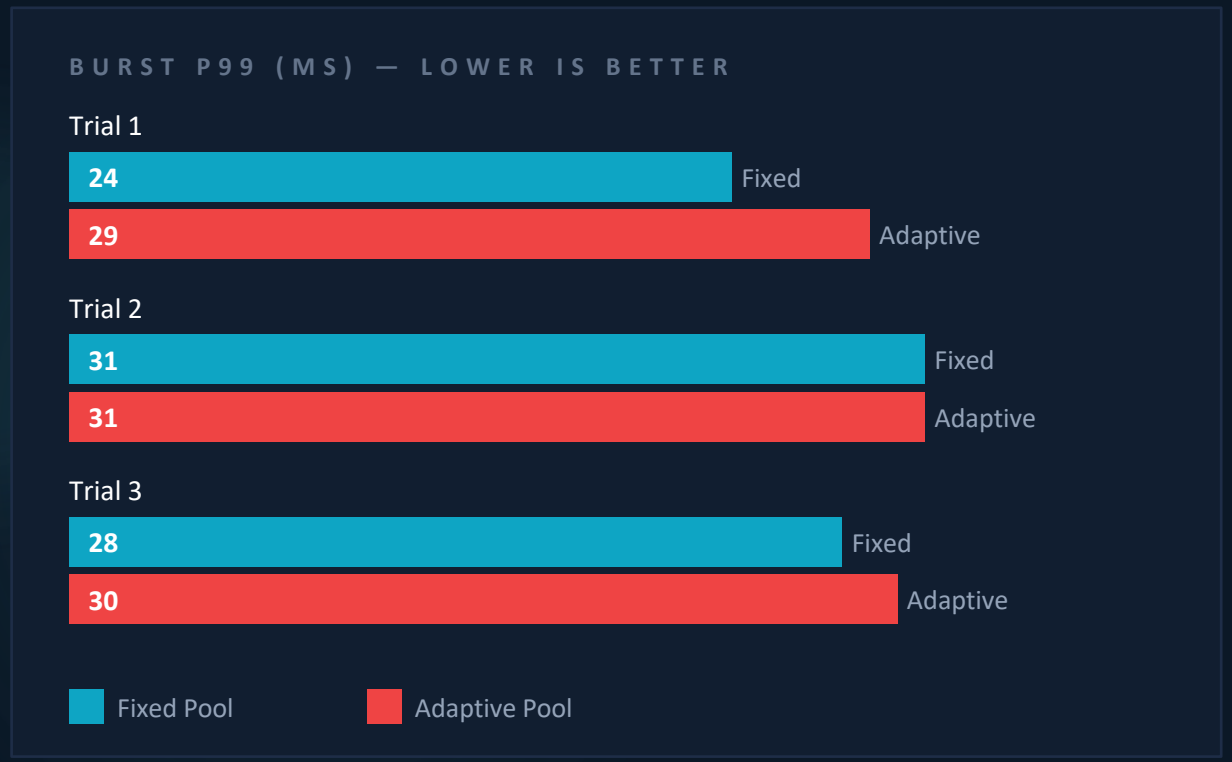
Noisy

Jumps 5× at c64+. Queue depth variance + OS scheduler jitter. The outliers feel very different from the median.



Burst test — 3 cycles of c2 baseline → c100 spike — fixed pool

Did We Hit 30% P99 Reduction?



PREDICTED

≥ 30%

P99 reduction

ACTUAL

+8%

adaptive was slower

Why? Every Request Locks the Mutex.

every request

`submit(job)`

→

⚠ every time, even if no-op

`LOCK workers_mutex`

`compute_target()`

→

scale? usually: no

`push to queue`

~6,700

mutex locks/sec just to ask

"should I scale?"

Fixed pool:

0 locks

after startup

What We Can't Claim

Same Machine

client + server compete for CPU. Benchmarks on the same host undercount true server latency.

ab omits slow reqs

Apache Bench drops timed-out requests from results. True P99 is likely worse than reported.

No Keep-Alive

Every request pays a full TCP handshake. Adaptive gains are buried in handshake noise.

Sub-ms Handler

Our handler is too fast. Adaptive scaling only helps when requests are long enough for new threads to matter.

Negative results are still results — as long as you know what constrained you.

One Change. Likely Flips the Result.

NOW

submit()

LOCK mutex

compute_target()

push to queue

× every single request

THE FIX

submit()

↓

push to queue only

Monitor Thread

checks every 100ms

scales if needed

✓ Zero mutex cost on hot path