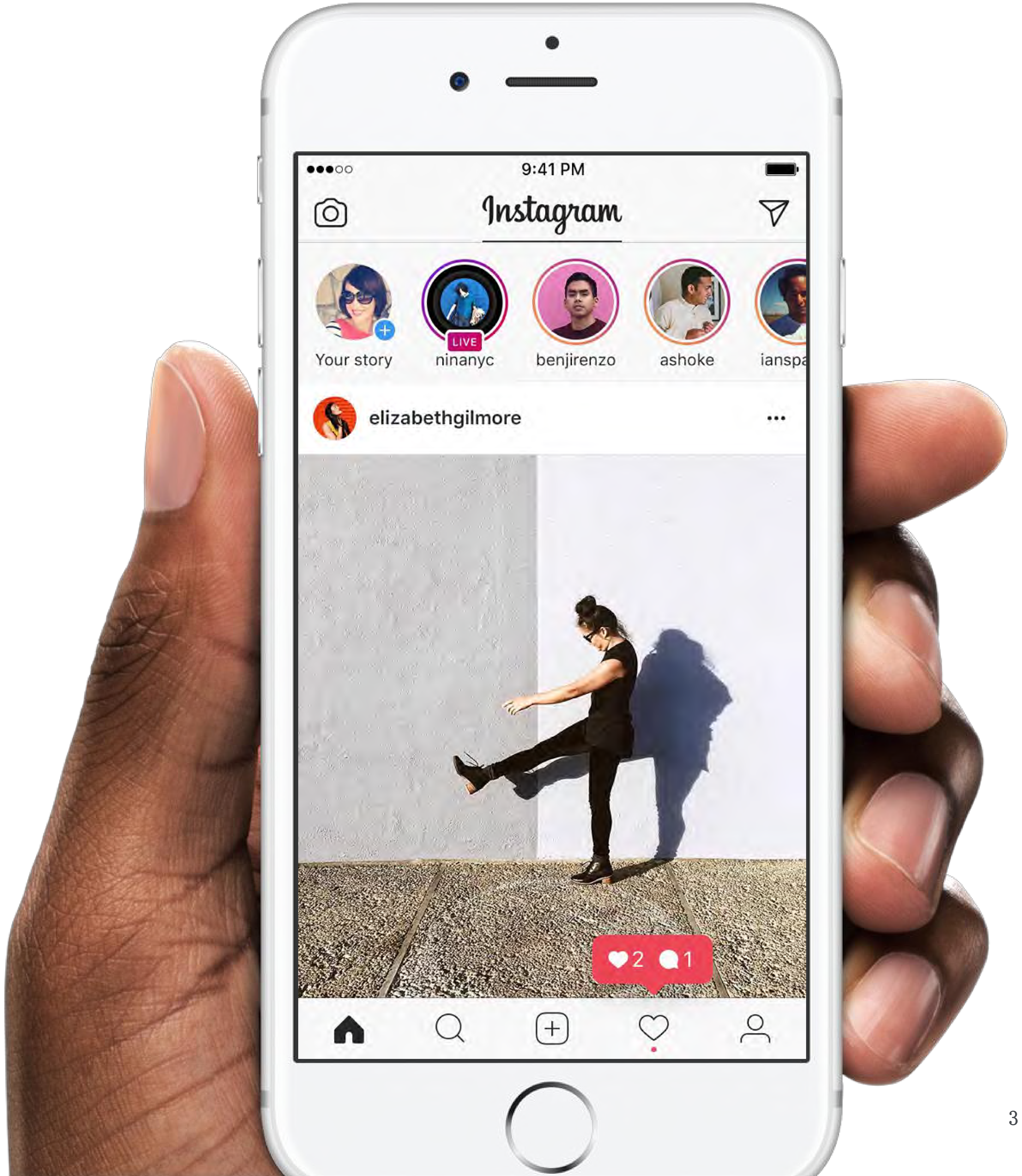
WEB SERVICE EFFICIENCY AT INSTAGRAM

2017 QCon Beijing

Hao Chen 陈昊
Instagram Infrastructure
chenh@fb.com

INSTAGRAM

- A community where people capture and share world’ s moments and tell their stories.
- More than 600M* people are using our service every month.



** Q4 Facebook earnings call, Feb 2017*

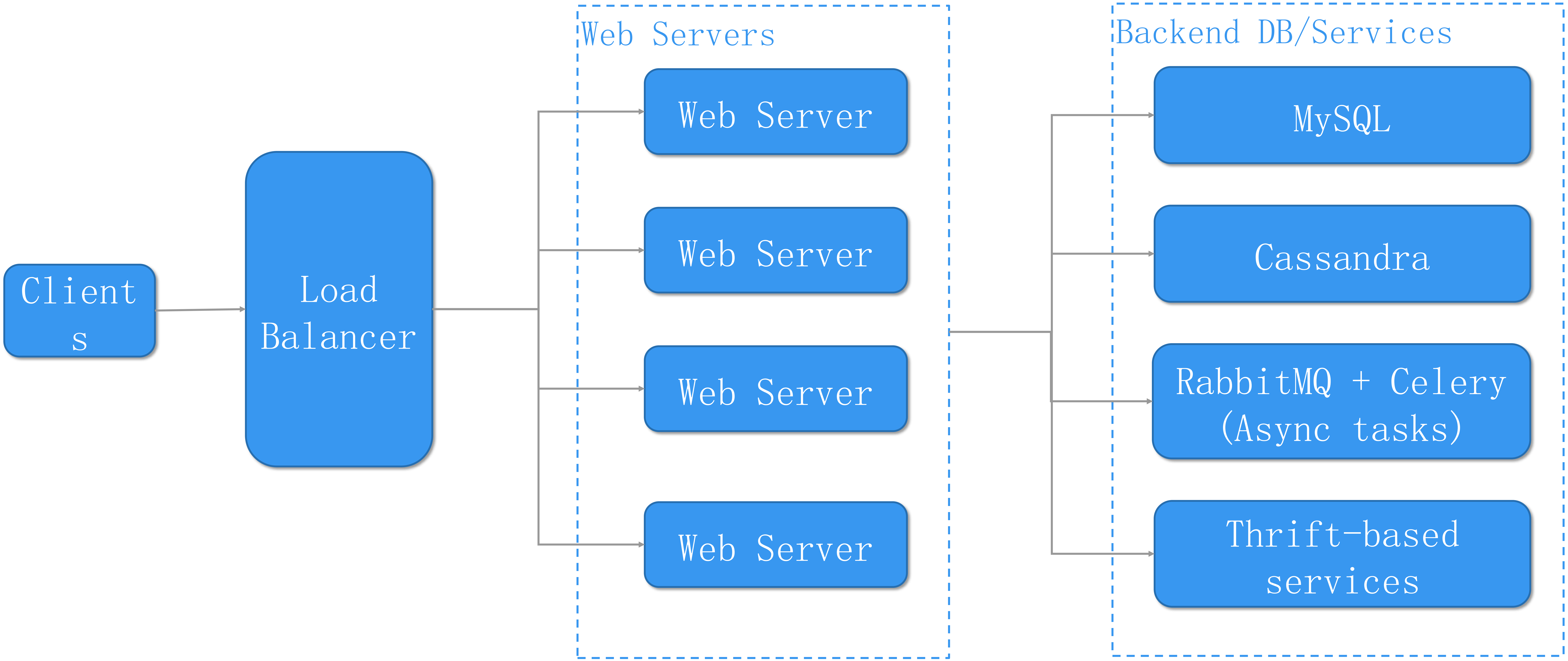
AGENDA

1 Overview

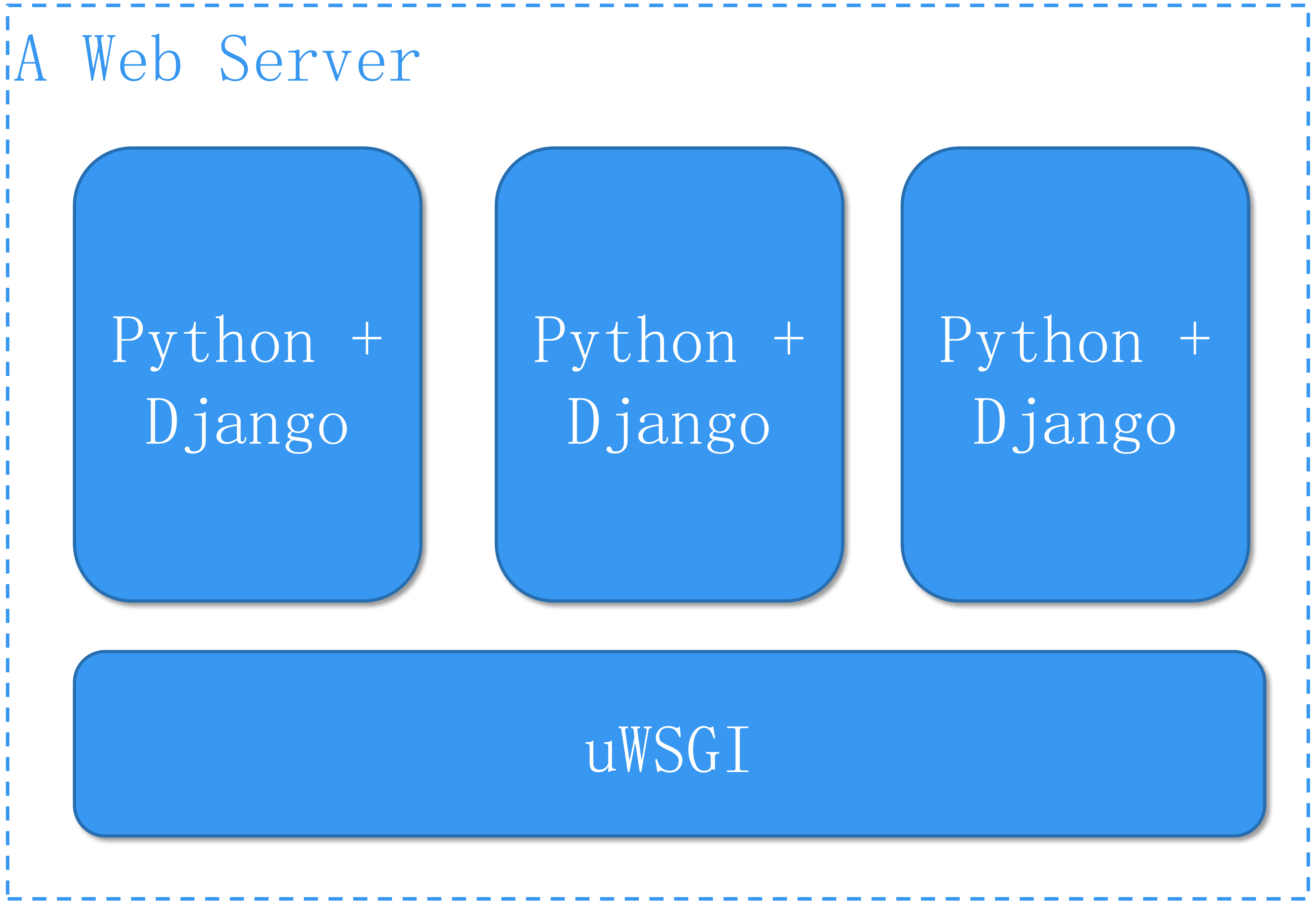
2 Efficiency Tooling

3 Optimization Case Study

IG ARCHITECTURE OVERVIEW

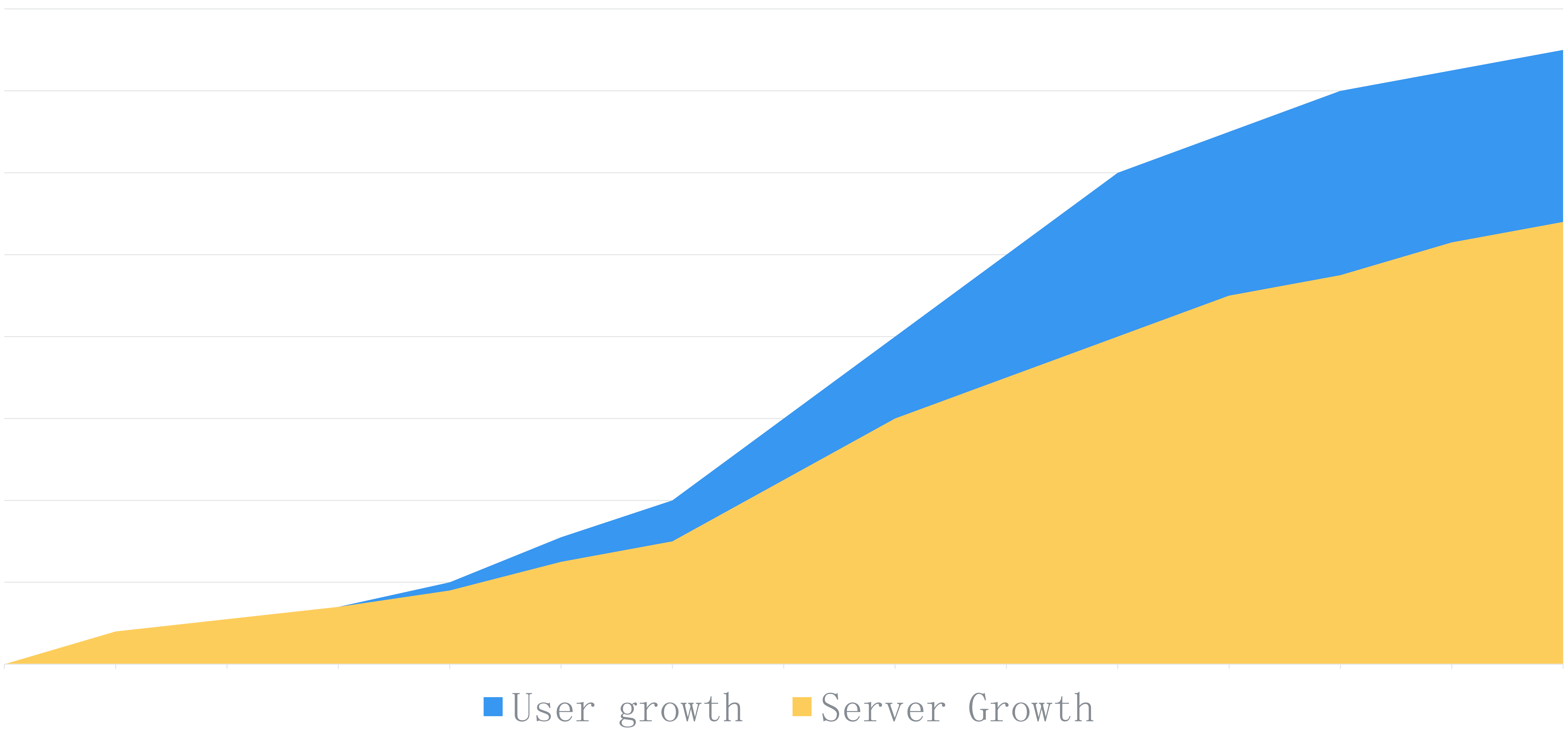


WHAT’S INSIDE A WEB SERVER?



WHY EFFICIENCY

- Servers and datacenter power are not free



- Serve as many users as possible with one server

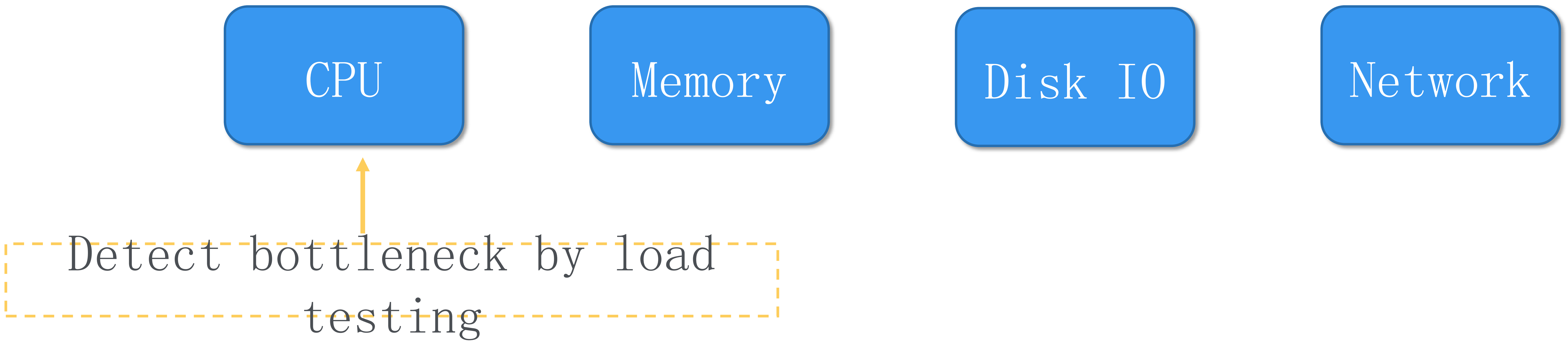
WHY EFFICIENCY (CONT’ D)

- Capacity utilization awareness
- Disaster readiness
- Capacity estimation for new products

DEFINING EFFICIENCY

Choose the target

Physical Resource Restrictions



QUANTIFYING EFFICIENCY

Choose the metric

- CPU Time
  - × Affected by CPU models
  - × Affected by runtime CPU load

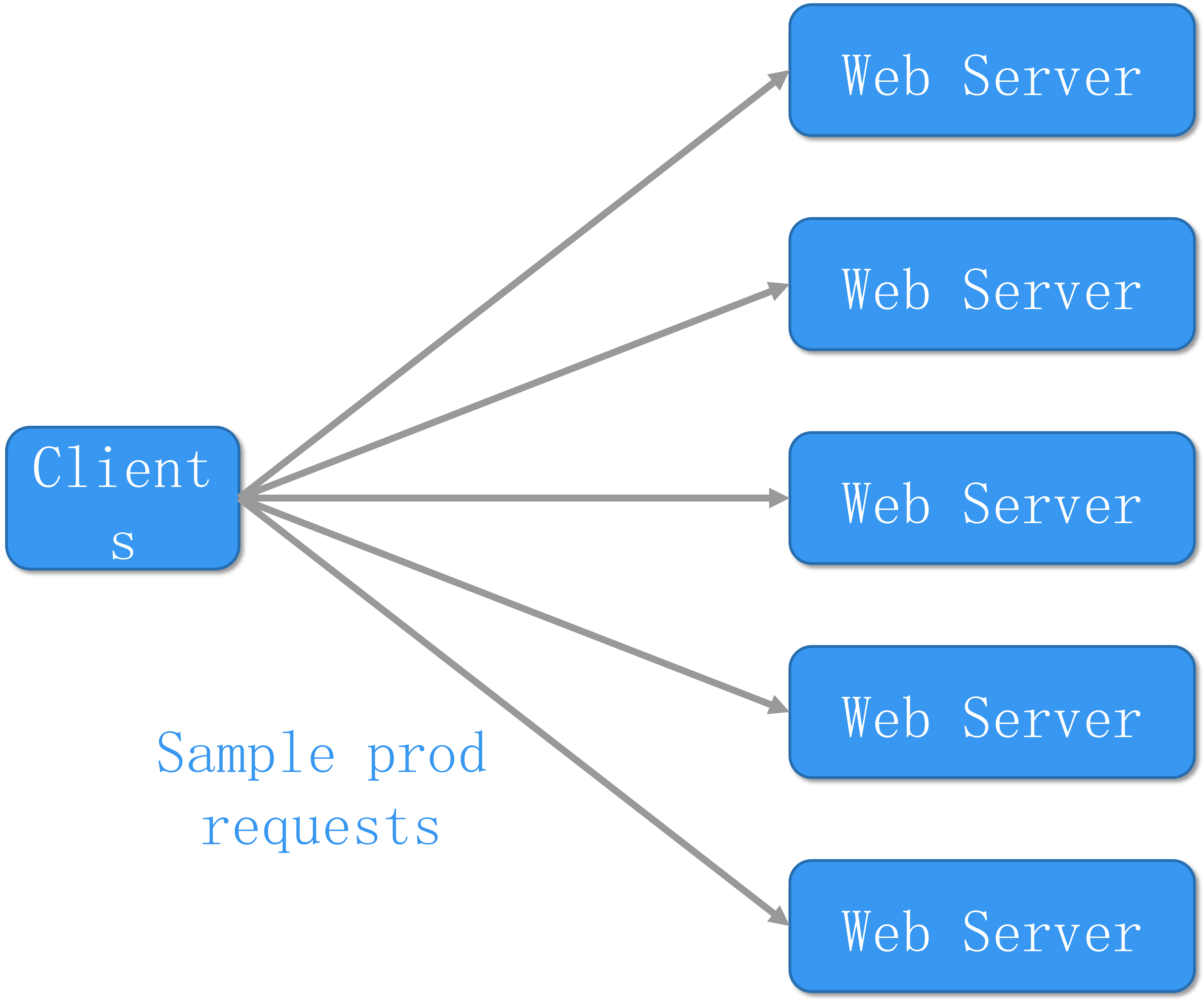
- CPU Instructions
  - ✓ Stable regardless of runtime environments
  - ✓ Measure via hardware counters on Linux

EFFICIENCY TOOLING

DETECTING REGRESSIONS

Efficiency Regression: Use more CPU instr to serve a request

DYNOSTATS

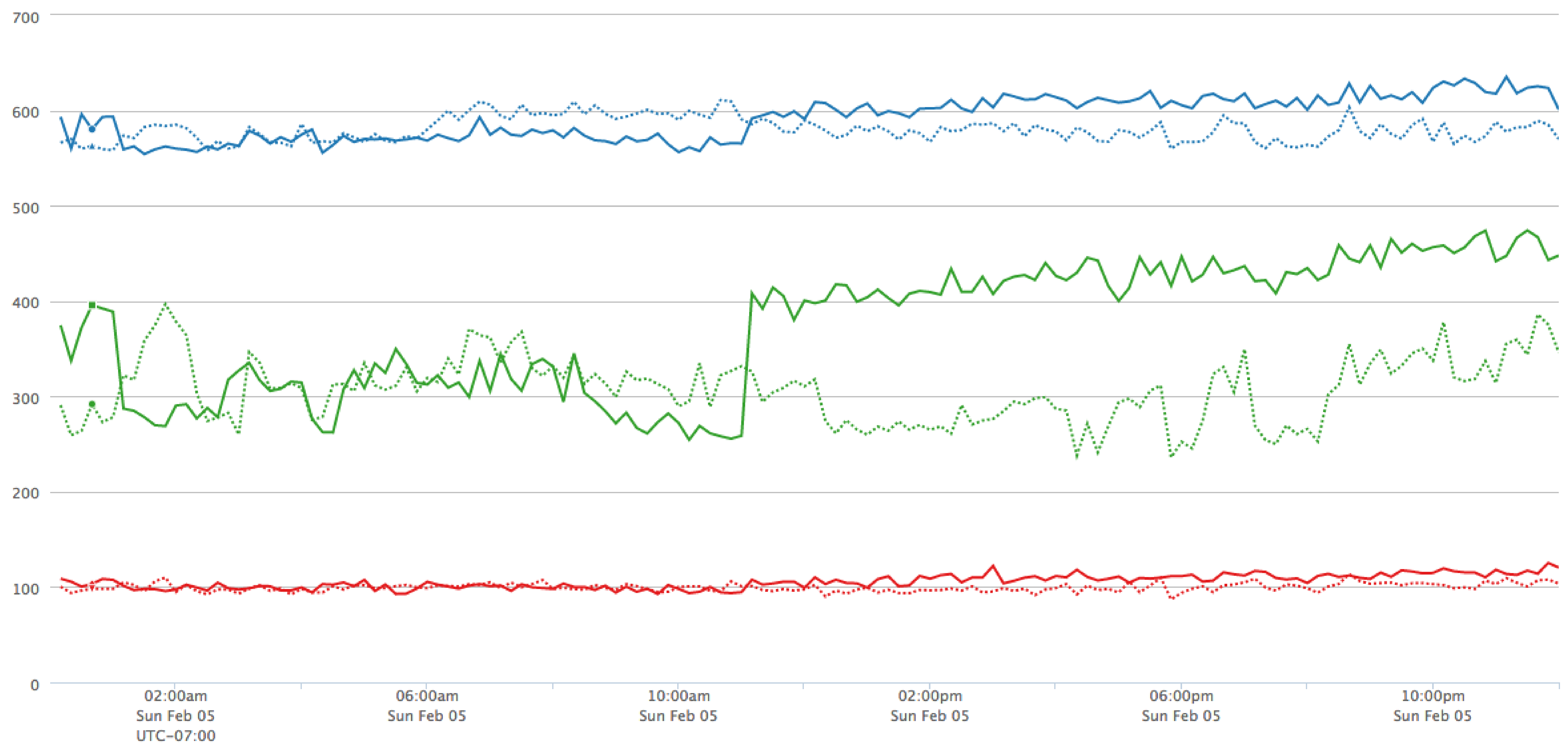


- Perf metrics
  - CPU Instr
  - Memory
  - Latency
  - Number and time of backend calls
  - …

- Metadata for aggregation
  - Endpoint name
  - Server/cluster name
  - Client platform (iOS/Android) and version
  - Configuration parameter
  - …

WHY DYNOSTATS?

- Detect regressions
  - When?
  - How much?
  - Which endpoint(s)?



- Monitor with Cron jobs
  - Fire alerts to on-call

TRIAGING REGRESSIONS

WHAT TO DO WITH REGRESSIONS?

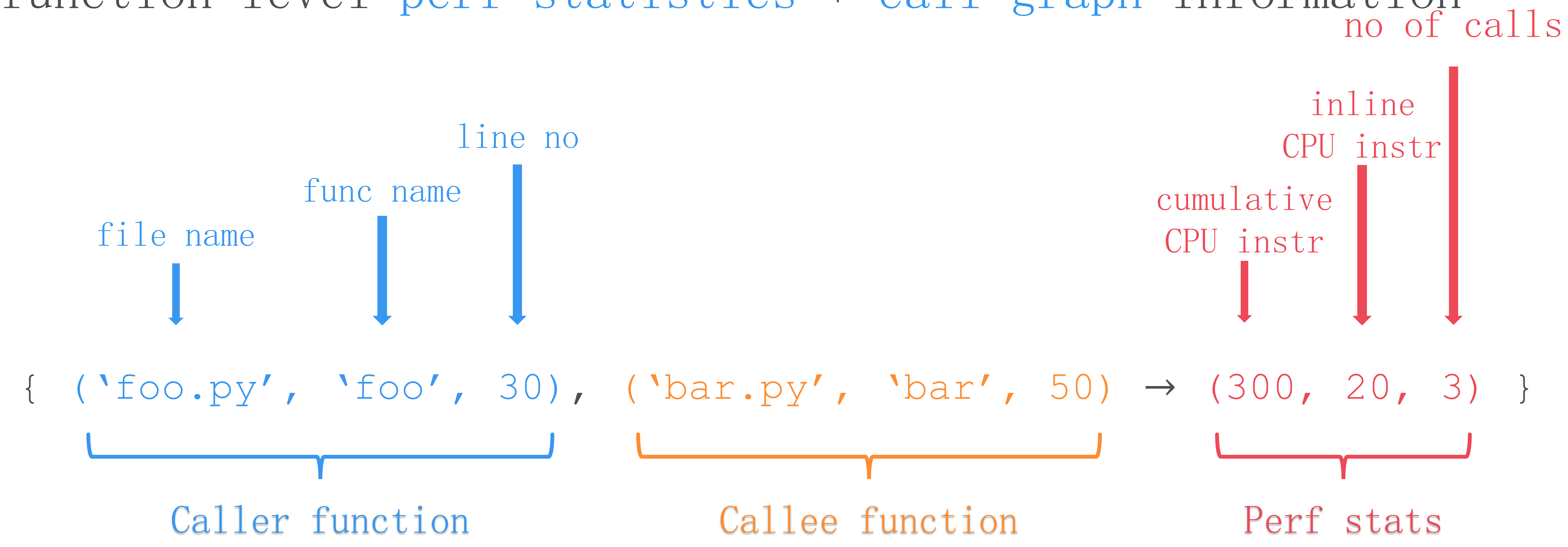
- Who introduced this regression?
  - Inefficient new code?
  - Configuration changes?

× Problem: Dynostats only has request-level metrics

✓ Solution: Function-level perf measurement

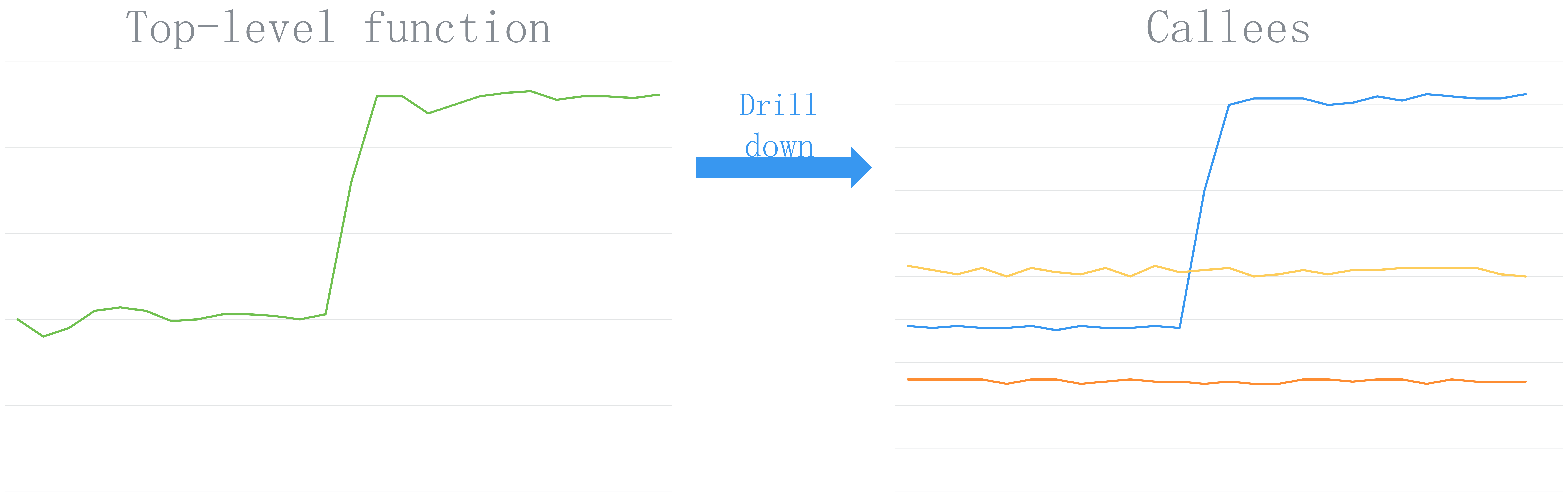
CPROFILE

- Python’ s built-in profiling tool
  - function-level perf statistics + call graph information



- Only enabled for a small subset of prod requests (because of overhead)

ROOT CAUSE REGRESSION



FINDING EXISTING BOTTLENECKS

- List most expensive functio

Function	Hits	Samples	Calibrated Cumulative CPU	Calibrated Self CPU [illegible]	Ncalls [illegible]
explore	3.42 M (0.3%)	9.86 K (0.3%)	3,192,384,698 (8.7%)	1,912,666 (0.2%)	3.42 M
search	6.30 M (0.5%)	18.2 K (0.5%)	3,078,815,091 (8.4%)	223,107 (0.0%)	7.35 M
__init__	2.63 M (0.2%)	7.58 K (0.2%)	2,180,869,044 (5.9%)	360,078 (0.0%)	2.63 M

  - Add a cache?
  - Optimize algorithm?
  - Re-write in C++?

- Example: 6.5% global CPU was used by imports
  - 4.2% saved by just removing in-function imports in hot functions

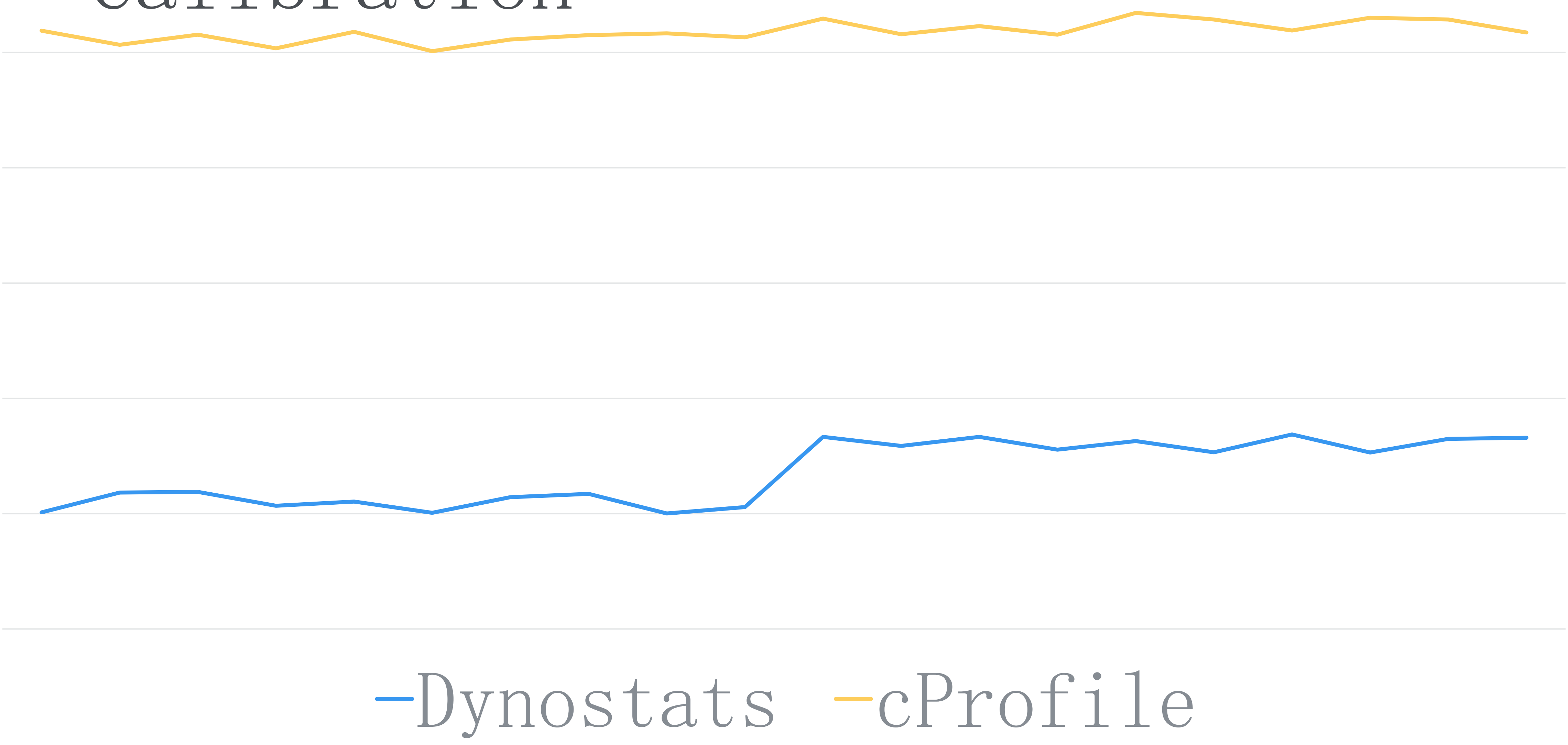
```
from a.b import func          import a
func()               →        a.b.func()
```

work around circular imports

OUT-OF-BOX CPROFILE ISN'T PERFECT

Our customizations to cProfile

- Doesn't distinguish decorator functions
- Hard to identify a function
  - Add class name
- Huge overheads hide regression
  - Calibration



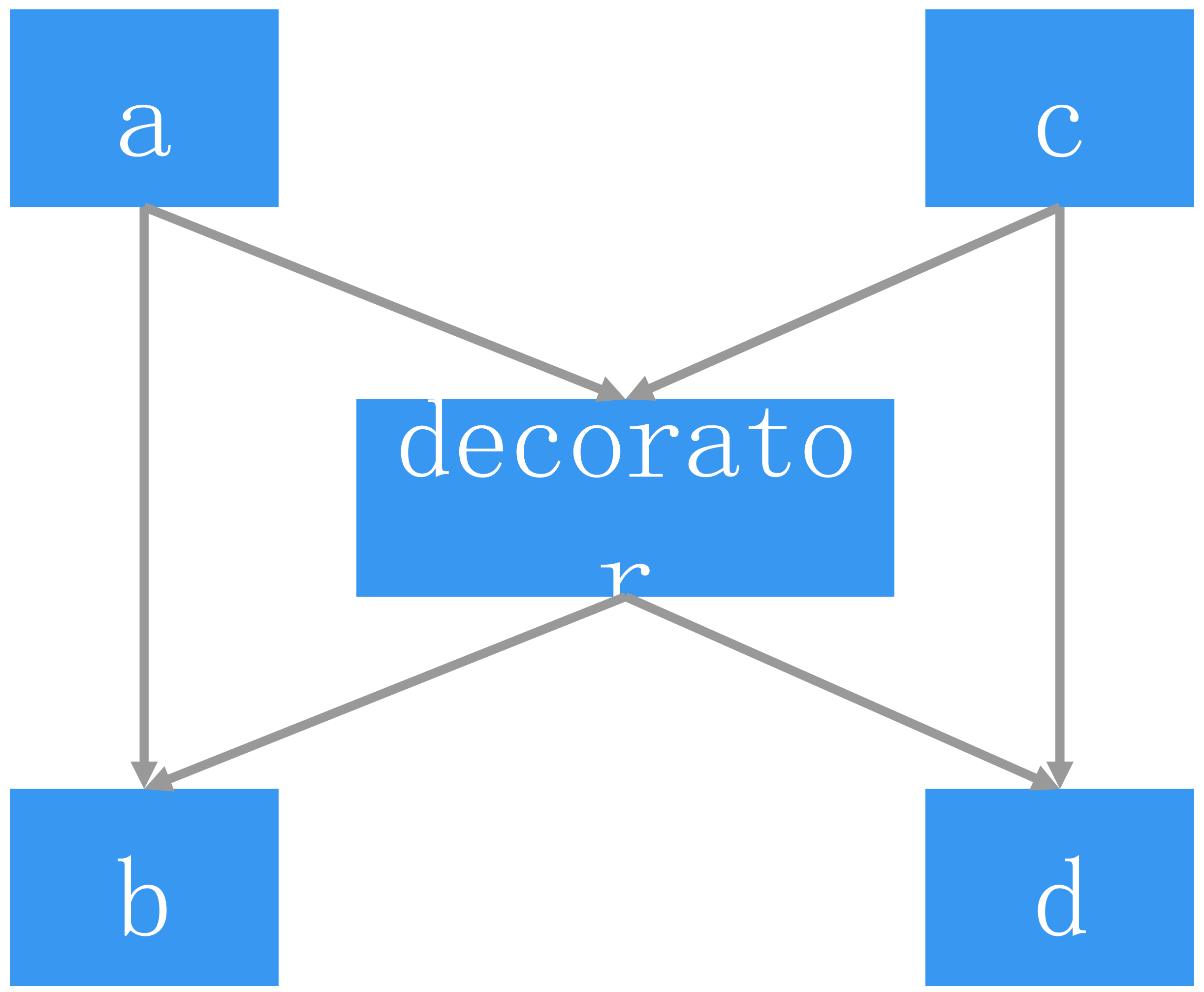
('foo.py', 'foo', 30)

```
def a():
    b()

@decorator
def b():
    pass
```

```
def c():
    d()

@decorator
def d():
    pass
```



PREVENTING REGRESSIONS

INSTALAB

- A traffic replay system
- Record prod requests and replay them in a controlled environment

WHY TRAFFIC REPLAY?

- Experiment changes without affecting production!
  - Detect efficiency regressions before new code lands
  - Catch elevated errors before new code lands
  - Reproduce failures

INSTALAB: EXAMPLE

An experiment input sample


```
{
  "workload":"instagram_server",
  "sides":[
    {
      "name":"a",
      "build_input":{
        "fbpkg_map":{
          "instagram.server":"3ede50ab5d80455f8901a73c547cdbc2"
        }
      }
    },
    {
      "name":"b",
      "build_input":{
        "fbpkg_map":{
          "instagram.server":"54c91e4bcc6c48158efe7ab2b8f06a17"
        }
      }
    }
  ]
}
```


Metric	(A) Value	(B) Value	(B) Delta	(B) Standard-Error	(B) P-Value (%)
treadmill.treadmill.response_500.sum.60	291	300	+8.91 (+3.0%)	+/- 6.79 (2.4%)	23.54
treadmill.treadmill.GET.sum.60	24.1	24.9	+0.732 (+2.2%)	+/- 2.71 (11.4%)	80.31
django-windtunnel.cpu_instr.feed.api.views.timeline	178 M	177 M	-916 K (-0.5%)	+/- 87 K (0.0%)	0.00
django-windtunnel.cpu_instr.igstats.views.health_check_queue	11.4 M	11.4 M	+61 K (+0.5%)	+/- 172 K (1.5%)	72.47
django-windtunnel.tw.mem.rss_bytes	7.74 B	7.68 B	-60.6 M (-0.8%)	+/- 83.3 M (1.1%)	47.22
django-windtunnel.tw.cpu.user	114	117	+3.05 (+2.7%)	+/- 0.812 (0.7%)	0.04

INSTALAB: CHALLENGE

- Problem: avoid writes to prod data

- Solution: intercept requests
  - Monkey patch functions
  - Drop writes or fake responses
  - Attach “don’t log” metadata to requests

SECTION RECAP

RECAP

- Detect: Dynostats
- Triage: cProfile
- Prevent: InstaLab

- Wins
  - Saved >70% global CPU in Q1 2017
  - Launched new major features without any capacity issue

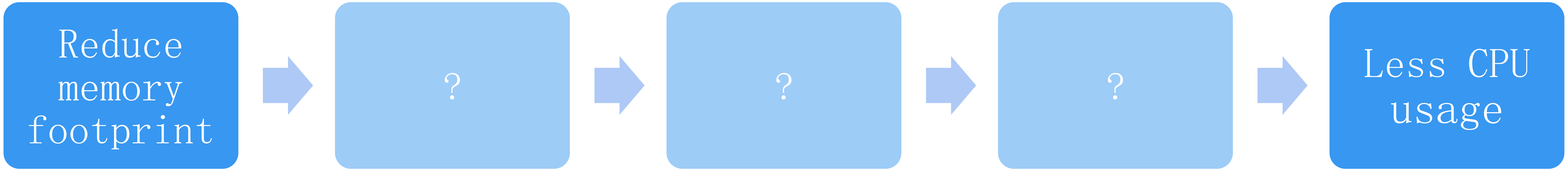
TAKEAWAYS

- Profile, profile & profile
- Caches fix most regressions
- Don’t do more than you need

OPTIMIZATION CASE STUDY

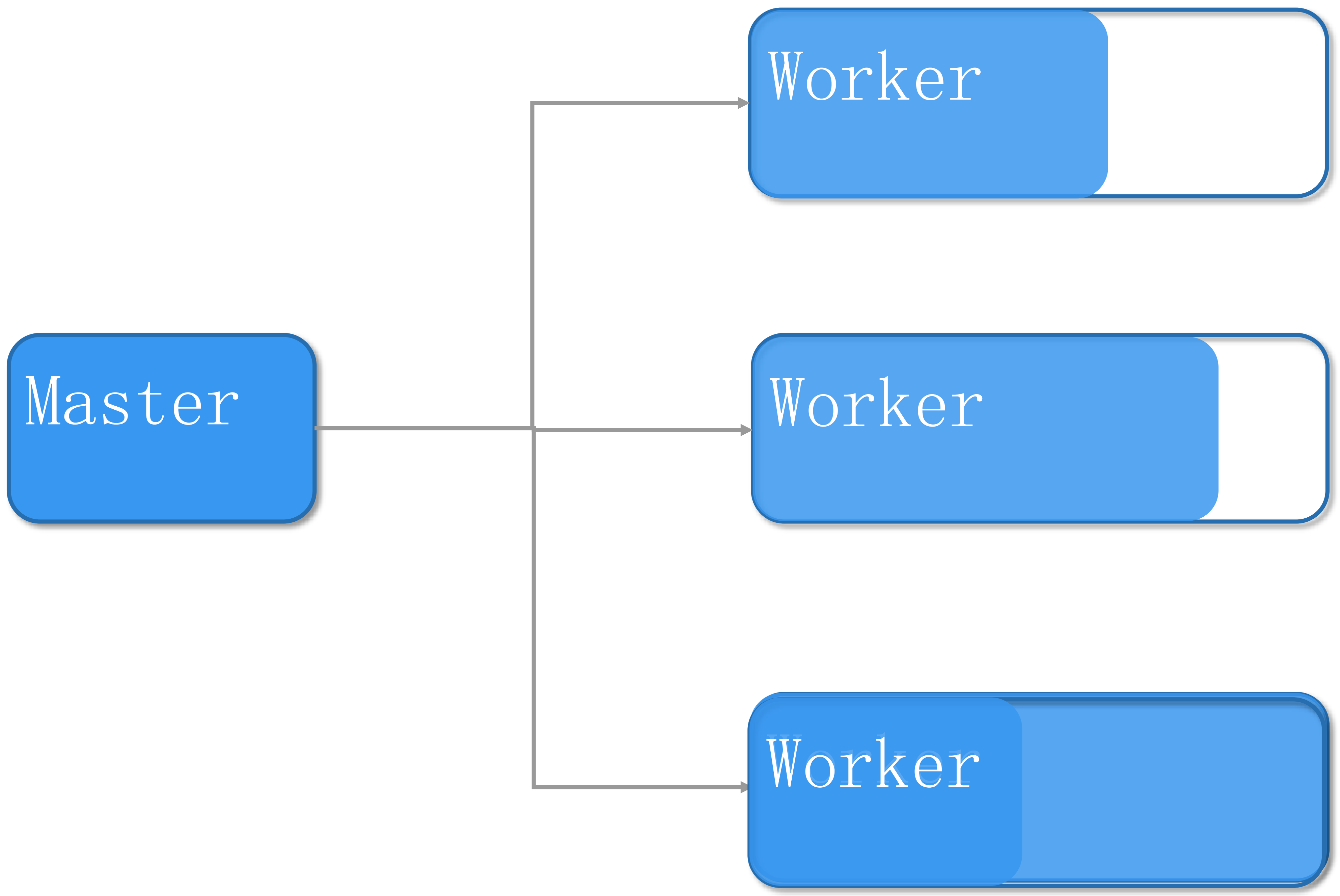
SHARED MEMORY

WHY SHARED MEMORY?



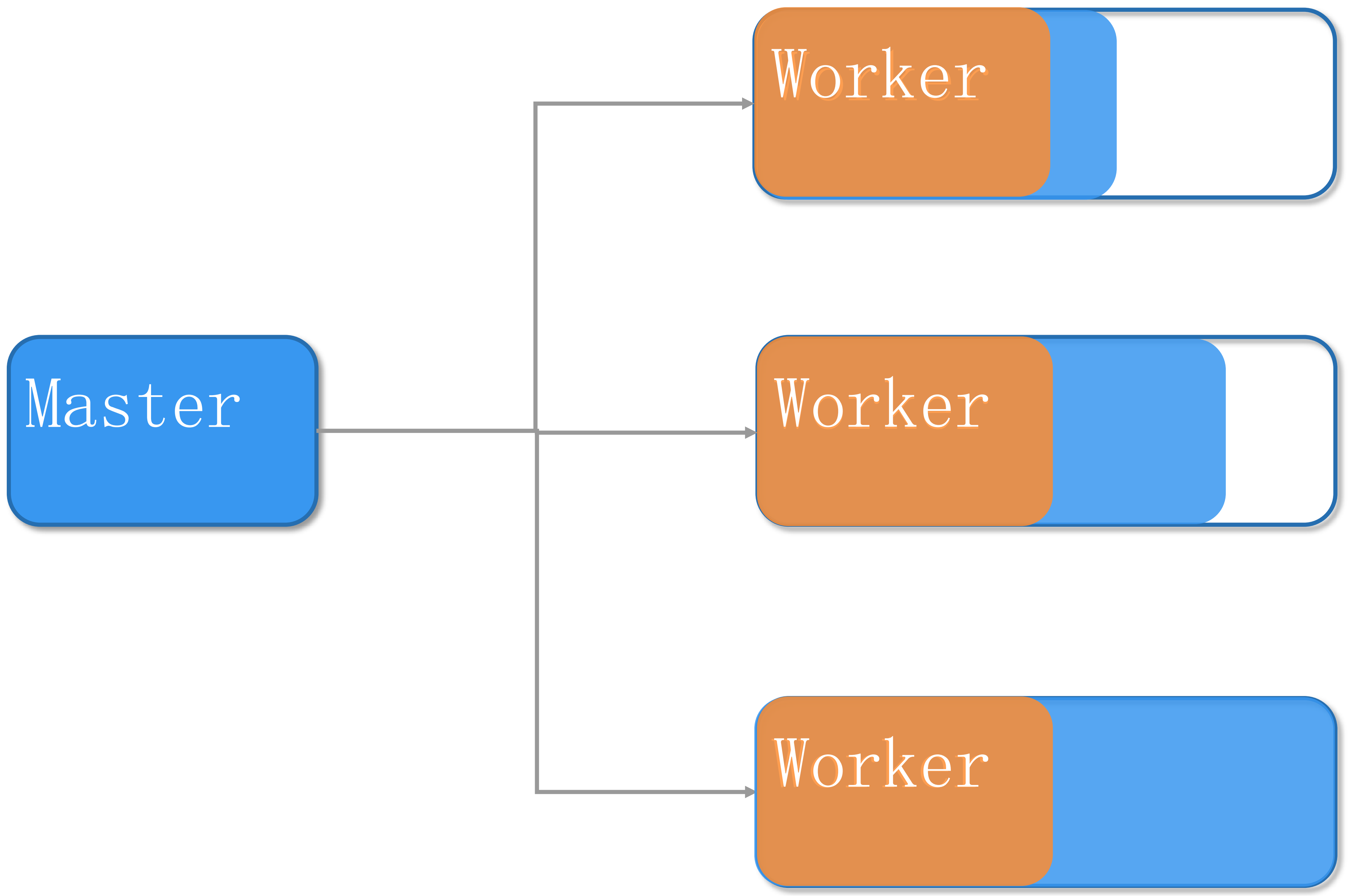
MASTER-WORKER MODEL

- Multi-process (because of GIL)
  - Worker handles requests
  - Master respawns worker when it exceeds memory limit



MASTER-WORKER MODEL (CONT’ D)

- Problem
  - Worker processes don’ t share memory with each other
  - Large in-memory configurations duplicated in each worker



WHY REDUCE MEMORY FOOTPRINT?

OPTIONS COMPARISON

Remove configs from workers' private memory

- Local in-memory DB (e.g. MC/Redis)
  - × Efficiency overhead: data copy via sockets
  - × Maintenance overhead
- Shared memory
  - ✓ Supported by uWSGI (uWSGI cache)
  - ✓ Simple key-value-style API
  - ✓ Memory allocated in master, shared by all workers
  - ✓ Tiny overhead (mmap)

```
uwsgi.cache_get(key, cache_name)
uwsgi.cache_update(key, value, expires, cache_name)
uwsgi.cache_keys(cache_name)
uwsgi.cache_clear(cache_name)
```

WINS

- Respawn rate: 58%
- Per-request memory growth: 65.03%
- Per-request CPU instr: 5.75%

A PITFALL

- Heavy reads, rare writes
  - read/write lock (pthread_rwlock_t)

- Issue: occasional deadlock in production
  - only 1~2 times per day among the whole fleet
  - very difficult to reproduce

A PITFALL (CONT’ D)

- Root cause: R/W lock
  - Created on OS level
  - Not released when worker killed
  - uWSGI’ s deadlock detector is buggy
    - only release the last reader
- Solution: Semaphore
  - uWSGI option: `lock-engine = ipcsem`
  - Negligible perf difference compared with r/w lock

- Takeaway: old, simple and reliable techniques are more preferable than the new and fancy ones

CYTHONIZATION

EXPENSIVE FUNCTIONS?

- Implement expensive functions in C++?
  - × Massive code changes
  - × New bugs
  - × Hard to measure gain before migrating everything

CYTHON IS YOUR FRIEND

- Cython is a Python-to-C compiler
  - write code in Python-like syntax
  - run code with C-like performance

✓ Compile Python code without changes

✓ Call back and forth between C and Python functions

✓ Static type declarations

  - Any C/C++ types: int, double, pointer, struct, union, STL

CYTHON WORKFLOW

1. Detect expensive modules (from profiling data)
   - Low-level, CPU intensive, Relatively stable
2. Compile it
3. Add static types

STATIC TYPES EXAMPLE

```
def f(x):
    return x * x

def g(n):
    result = 0
    for i in range(N):
        result += f(i)
    return result
```

```
cdef long f(int x):
    return x * x

def g(int n):
    cdef:
        long result
        int i
    result = 0
    for i in range(N):
        result += f(i)
    return result
```

150x Faster!

```
for(i=0; i<N; i++)
   result+= f(i)
```

CYTHON WORKFLOW

1. Detect expensive modules (from profiling data)
   - Low-level, CPU intensive, Relatively stable
2. Compile it
3. Add static types
4. [Optional] Apply additional optimizations
   - Low-level features: STL; Raw pointers; Pure C code

✓ Minor code changes

✓ Progressive optimization

CYTHON: CHALLENGES

- Slow compilation
- Incompatibilities
- Debugging and profiling tools support

CYTHON: RECAP

- 10-ish modules converted
- 30% global CPU Win

Eng blog: https://engineering.instagram.com