>_

DevOps Tricks

Complete Reference · Beginner → Advanced

Do Cao Hieu

Table of Contents

PART I — TOOL REFERENCE

01 Linux & Bash Tricks — Từ Cơ Bản Đến Nâng Cao 5

02 Git Tricks — Từ Cơ Bản Đến Nâng Cao 16

03 Docker Tricks: Từ Beginner đến Advanced 27

04 Kubernetes Tricks: Từ Beginner đến Advanced 38

05 Nginx Tricks — Từ Cơ Bản Đến Nâng Cao 53

06 Traefik Tricks — Từ Cơ Bản Đến Nâng Cao 67

07 Redis Tricks — Beginner to Advanced 82

08 PostgreSQL Tricks — Beginner to Advanced 93

09 Terraform Tricks — Beginner to Advanced 108

10 Ansible Tricks — Beginner to Advanced 121

11 Prometheus & Grafana — DevOps Tricks 137

12 GitHub Actions — DevOps Tricks 150

13 GitLab CI/CD Tricks — Beginner to Advanced 164

14 Networking & DNS Tricks — DevOps từ Beginner đến Advanced 177

PART II — SERVER OPTIMIZATION

15 Server Optimization Architecture — Toàn cảnh tối ưu từ bit đến cluster 193

16 Memory-Efficient Data Structures — Xử lý hàng triệu entity với vài KB RAM 198

17 Kernel & OS Tuning — Biến VPS 4GB thành cỗ máy chịu 1M connections 217

18 Connection Pooling & Management — Multiplexing 1M users qua 100 connections 228

19 Caching Strategies — Giảm 99% load bằng multi-layer cache 242

20 Database Optimization at Scale — PostgreSQL/MySQL chịu 100K queries/sec 255

21 Application-Level Patterns — Code patterns giúp 1 process xử lý 100K req/sec 272

22 Async Processing & Message Queues — Absorb traffic spikes without scaling 288

23 Load Balancing & Reverse Proxy — Distribute 1M connections across small servers. 305

PART I

Tool Reference

14 công cụ — tricks từ cơ bản tới nâng cao.

01

Linux & Bash Tricks — Từ Cơ Bản Đến Nâng Cao

> Giải thích bằng tiếng Việt, lệnh bằng tiếng Anh. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

1. Process Management

B Xem tiến trình đang chạy với ps

```
ps aux
# USER  PID  %CPU %MEM  VSZ   RSS  TTY STAT START   TIME COMMAND
# root    1   0.0  0.1 169444 10240 ?   Ss  May01   0:05 /sbin/init

ps aux | grep nginx          # lọc theo tên
ps -ef --forest              # hiển thị dạng cây cha-con
ps -p 1234 -o pid,ppid,cmd,%cpu,%mem  # cột tuỳ chỉnh
```

B Giám sát realtime với top / htop

```
top -u www-data              # chỉ xem process của user cụ thể
top -d 1                     # refresh mỗi 1 giây
# Phím tắt trong top: P=sort CPU, M=sort MEM, k=kill, q=quit

htop -d 5 -u deploy          # htop: refresh 0.5s, filter user
# htop cần cài: apt install htop
```

B Kill tiến trình

```
kill 1234                    # gửi SIGTERM (graceful)
kill -9 1234                 # gửi SIGKILL (force, không cleanup)
kill -HUP 1234               # SIGHUP: reload config (nginx, sshd)
pkill nginx                  # kill theo tên
pkill -u deploy              # kill toàn bộ process của user
killall -9 java              # kill tất cả process tên java

# ! kill -9 không cho process dọn dẹp file tạm, socket — dùng SIGTERM trước
```

I Chạy ngầm với nohup và disown

```
nohup ./long-script.sh > output.log 2>&1 &
# nohup: bỏ qua SIGHUP khi terminal đóng
# output mặc định vào nohup.out nếu không redirect

# Đã chạy rồi mới muốn tách ra:
./long-script.sh &           # chạy background
jobs                         # xem job number
disown %1                    # tách job 1 khỏi shell, không nhận SIGHUP

# ! disown không redirect stdout — process vẫn chết nếu terminal đóng và nó cố ghi vào tty
```

I systemd — quản lý service

```
systemctl start nginx
systemctl stop nginx
systemctl restart nginx
systemctl reload nginx       # reload config không restart
systemctl enable nginx       # bật autostart
systemctl disable nginx
systemctl status nginx       # xem trạng thái + log gần nhất

systemctl list-units --type=service --state=running   # liệt kê service đang chạy
systemctl list-units --failed                         # service bị lỗi
systemctl --user start myapp.service                  # service của user (không cần root)

# Edit service file:
systemctl edit nginx         # override không sửa file gốc (drop-in)
systemctl edit --full nginx  # sửa file gốc
```

I journalctl — đọc log hệ thống

```
journalctl -u nginx                        # log của service nginx
journalctl -u nginx -f                     # follow realtime
journalctl -u nginx --since "1 hour ago"
journalctl -u nginx --since "2024-01-01" --until "2024-01-02"
journalctl -p err -b                       # chỉ lỗi, từ boot hiện tại
journalctl -b -1                           # log boot trước đó
journalctl --disk-usage                    # xem dung lượng log
journalctl --vacuum-time=7d               # xoá log cũ hơn 7 ngày
journalctl -o json-pretty -u nginx | head  # output JSON
```

A Giới hạn tài nguyên với cgroups / systemd

```
# Chạy lệnh với giới hạn CPU và RAM:
systemd-run --scope -p MemoryLimit=512M -p CPUQuota=50% ./heavy-job.sh

# Xem cgroup của process:
cat /proc/1234/cgroup

# ! MemoryLimit dùng MB/GB, CPUQuota là % của 1 core (200% = 2 cores)
```

---

2. Filesystem

B Tìm file với find

```
find /var/log -name "*.log" -mtime -7       # log mới hơn 7 ngày
find . -type f -size +100M                  # file lớn hơn 100MB
find /tmp -type f -mmin +60 -delete         # xoá file cũ hơn 60 phút
find . -name "*.py" -exec grep -l "import os" {} \;   # tìm file chứa chuỗi
find . -perm -4000                          # tìm file SUID

# ! -delete phải đi sau các điều kiện lọc, không thể undo
```

I fd — thay thế find nhanh hơn

```
# Cài: apt install fd-find hoặc cargo install fd-find
fd '\.log$' /var/log             # tìm file .log
fd -t f -S +100m .               # file lớn hơn 100MB (-S = size; -s là case-sensitive)
fd -e py --exec grep -l "import os"  # tìm và exec

# fd nhanh hơn find vì dùng multiple threads và bỏ qua .gitignore mặc định
```

B Phân tích dung lượng với du / ncdu

```
du -sh /var/log                  # tổng dung lượng thư mục
du -sh /* | sort -rh | head -20  # top thư mục lớn nhất
du -ah --max-depth=2 /var        # chi tiết 2 cấp

# ncdu: interactive TUI (cài: apt install ncdu)
ncdu /var
# Điều hướng bằng arrow keys, d=delete, q=quit
```

I lsof — xem file đang mở

```
lsof -i :80                      # process nào đang dùng port 80
lsof -u www-data                 # file đang mở bởi user
lsof +D /var/log                 # file trong thư mục đang mở
lsof -p 1234                     # file đang mở bởi PID
lsof | grep deleted              # file đã xoá nhưng vẫn bị giữ (chiếm disk)

# ! File deleted nhưng process vẫn giữ fd → disk không giải phóng cho đến khi process đóng
# Fix: lsof | grep deleted → restart process hoặc truncate: > /proc/PID/fd/FD
```

I inotifywait — theo dõi thay đổi filesystem

```
inotifywait -m -r -e modify,create,delete /etc/nginx/
# Chạy liên tục, in ra mỗi khi có thay đổi trong /etc/nginx/
# Output: /etc/nginx/ MODIFY nginx.conf

# Tự động reload khi config thay đổi:
while inotifywait -e close_write /etc/app/config.yml; do
  systemctl reload myapp
done
```

I xargs — pipe argument hiệu quả

```
find . -name "*.tmp" | xargs rm -f           # xoá file tìm được
find . -name "*.py" | xargs wc -l            # đếm dòng
cat hosts.txt | xargs -P 8 -I {} ping -c1 {} # ping song song 8 luồng
ls *.log | xargs -I {} gzip {}               # nén từng file

# ! Tên file có space sẽ bị bẻ gãy — dùng -print0 và -0:
find . -name "*.tmp" -print0 | xargs -0 rm -f
```

I rsync — đồng bộ file an toàn

```
rsync -avz /src/ user@host:/dst/             # sync với SSH
rsync -avz --delete /src/ /dst/              # xoá file không có ở src
rsync -avz --exclude='*.log' --exclude='.git' /src/ /dst/
rsync -avz --progress --bwlimit=10000 /large-file host:/path/  # giới hạn băng thông 10MB/s
rsync -n -avz /src/ /dst/                    # dry-run, không thực sự copy

# ! Dấu / cuối đường dẫn nguồn rất quan trọng:
# /src/  → copy nội dung bên trong src
# /src   → copy cả thư mục src
```

3. Text Processing

B grep — tìm kiếm văn bản

```
grep -r "error" /var/log/          # tìm đệ quy
grep -i "ERROR" file.log           # không phân biệt hoa thường
grep -n "pattern" file             # hiển thị số dòng
grep -v "DEBUG" app.log            # loại trừ dòng match
grep -E "error|warn|crit" app.log # regex mở rộng
grep -A3 -B3 "Exception" app.log  # 3 dòng trước/sau match
grep -c "404" access.log           # đếm số dòng match

# Tìm IP xuất hiện nhiều nhất:
grep -oE "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" access.log | sort | uniq -c | sort -rn
| head
```

I awk — xử lý cột

```
awk '{print $1, $4}' access.log              # in cột 1 và 4
awk -F: '{print $1}' /etc/passwd             # tách bằng :, in cột 1
awk '$3 > 100 {print $0}' data.txt           # lọc dòng có cột 3 > 100
awk '{sum += $5} END {print sum}' data.txt   # tính tổng cột 5
awk 'NR%2==0' file                           # chỉ dòng chẵn
awk '/START/,/END/' file                     # in từ dòng START đến END

# Thống kê HTTP status code:
awk '{print $9}' access.log | sort | uniq -c | sort -rn
```

I sed — chỉnh sửa stream

```
sed 's/old/new/g' file.txt                   # thay thế tất cả
sed -i 's/old/new/g' file.txt                # sửa trực tiếp file
sed -i.bak 's/old/new/g' file.txt            # sửa + backup .bak
sed -n '10,20p' file                         # in dòng 10-20
sed '/^#/d' config.txt                       # xoá dòng bắt đầu bằng #
sed -i 's/password=.*/password=REDACTED/' config.ini

# ! sed -i trên macOS cần argument sau -i: sed -i '' 's/old/new/' file
```

B cut, sort, uniq

```
cut -d: -f1,3 /etc/passwd                    # lấy cột 1 và 3 (delimiter :)
cut -c1-10 file                              # lấy ký tự 1-10 mỗi dòng

sort -k2 -n file                             # sort theo cột 2, số
sort -t: -k3 -n /etc/passwd                  # sort theo UID
sort -rn numbers.txt                         # sort ngược, numeric
sort -u file                                 # sort + loại duplicate

uniq -c file                                 # đếm số lần lặp
uniq -d file                                 # chỉ in dòng duplicate
```

I jq — xử lý JSON

```
cat data.json | jq '.'                        # pretty print
jq '.users[].name' data.json                  # lấy field
jq '.[] | select(.age > 25)' data.json        # filter
jq '.[] | {name, email}' data.json            # project fields
jq -r '.name' data.json                       # raw output (không có quotes)
curl -s api.example.com/users | jq '.[0].email'   # từ API
```

```
# Tổng hợp:
cat sales.json | jq '[.[] | .amount] | add'        # tổng tất cả amount
```

ℹ yq — xử lý YAML (như jq cho YAML)

```
yq '.services.nginx.image' docker-compose.yml
yq -i '.replicas = 3' deployment.yaml              # sửa trực tiếp
yq eval-all '. as $item ireduce ({}; . * $item)' *.yaml  # merge nhiều file
```

---

4. Networking

ℹ ss — xem socket/connection (thay netstat)

```
ss -tlnp                          # TCP listening, numeric, với process
ss -tunlp                         # TCP+UDP listening
ss -s                             # thống kê tổng quan
ss -tp state established          # kết nối đang established
ss 'dst 10.0.0.1'                 # kết nối đến host cụ thể
ss -o state time-wait             # connections đang TIME_WAIT

# ⚠ netstat deprecated trên nhiều distro, dùng ss thay thế
```

ℹ curl — HTTP client mạnh mẽ

```
curl -I https://example.com                  # chỉ headers
curl -v https://example.com                  # verbose, xem TLS handshake
curl -o /dev/null -s -w "%{http_code} %{time_total}\n" https://example.com  # check latency
curl -X POST -H "Content-Type: application/json" -d '{"key":"val"}' https://api.example.com
curl -u user:pass https://api.example.com    # basic auth
curl --retry 3 --retry-delay 5 https://api  # retry
curl -L https://example.com                  # follow redirects
curl -k https://self-signed.example.com      # bỏ qua SSL verify

# Upload file:
curl -F "file=@/path/to/file" https://upload.example.com
```

ℹ tcpdump — bắt gói tin

```
tcpdump -i eth0 port 80                      # bắt traffic port 80
tcpdump -i any host 10.0.0.1                 # traffic từ/đến IP
tcpdump -i eth0 -w capture.pcap              # lưu ra file (mở bằng Wireshark)
tcpdump -i eth0 -nn -s0 port 443            # -nn: không resolve DNS, -s0: full packet
tcpdump 'tcp[tcpflags] & tcp-syn != 0'       # bắt SYN packets

# ⚠ Cần root hoặc CAP_NET_RAW. Trên production dùng giới hạn thời gian: -G 60 -W 1
```

ℹ dig — DNS lookup

```
dig example.com                              # lookup A record
dig example.com MX                           # MX record
dig @8.8.8.8 example.com                     # query Google DNS
dig +short example.com                       # chỉ IP, ngắn gọn
dig +trace example.com                       # trace từng bước delegation
dig -x 1.2.3.4                              # reverse lookup (PTR)
dig example.com ANY                          # tất cả record types
```

I ip — quản lý network interface

```
ip addr show                                  # xem IP của các interface
ip route show                                 # bảng routing
ip route add 10.0.0.0/8 via 192.168.1.1      # thêm route
ip link set eth0 up/down                      # bật/tắt interface
ip neigh show                                 # ARP table
ip -s link show eth0                          # thống kê packets

# Xem bandwidth realtime:
watch -n1 'ip -s link show eth0'
```

A nmap — scan mạng

```
nmap -sV 192.168.1.0/24                      # scan subnet, detect version
nmap -p 22,80,443 host                        # scan port cụ thể
nmap -sU -p 53,161 host                      # UDP scan
nmap -O host                                  # OS detection
nmap --script vuln host                       # scan vulnerability
# ! Chỉ scan mạng bạn có quyền. Scan trái phép là vi phạm pháp luật.
```

---

5. Performance

I vmstat — thống kê memory, CPU, IO

```
vmstat 1 10                       # 10 lần, cách 1 giây
# r=run queue, b=blocked, si/so=swap in/out, bi/bo=block IO, us/sy/id/wa=CPU%

vmstat -s                         # summary memory
vmstat -d                         # disk statistics
```

I iostat — thống kê disk IO

```
iostat -x 1                       # extended stats, cách 1 giây
# %util: % thời gian disk bận. Gần 100% = disk là bottleneck
# await: thời gian chờ IO trung bình (ms)
# r/s, w/s: reads/writes per second

iostat -h                         # human-readable units
```

I sar — thu thập và báo cáo system activity

```
sar -u 1 5                        # CPU 5 lần cách 1s
sar -r 1 5                        # memory utilization
sar -n DEV 1 5                    # network by interface
sar -b 1 5                        # IO transfer rate

# Xem lịch sử hôm qua:
sar -u -f /var/log/sysstat/sa$(date -d yesterday +%d)
```

A strace — theo dõi syscall của process

```
strace -p 1234                    # attach vào process đang chạy
strace -p 1234 -e trace=network   # chỉ syscall network
strace -c ./program               # thống kê syscall (tổng thời gian, số lần)
strace -T ./program               # thời gian từng syscall
strace -o trace.txt ./program     # lưu ra file
```

```
# Debug "file not found":
strace -e trace=openat ./program 2>&1 | grep -i "no such"
```

A /proc filesystem

```
cat /proc/cpuinfo                # thông tin CPU
cat /proc/meminfo                # thông tin memory
cat /proc/1234/status            # trạng thái process
cat /proc/1234/maps              # memory map của process
cat /proc/1234/net/tcp           # TCP connections của process
cat /proc/sys/vm/swappiness      # swappiness value
echo 10 > /proc/sys/vm/swappiness  # thay đổi realtime (không persist reboot)

# Xem giới hạn file descriptor của process:
cat /proc/1234/limits
ls /proc/1234/fd | wc -l         # đếm fd đang mở
```

A perf — profiling CPU

```
perf top                         # giống top nhưng theo hàm/symbol
perf stat ./program              # thống kê hardware counter
perf record -g ./program         # ghi lại call graph
perf report                      # xem báo cáo sau perf record

# ❗ Cần kernel symbols: apt install linux-perf linux-tools-$(uname -r)
```

---

6. Security

B chmod/chown patterns

```
chmod 644 file                   # rw-r--r-- (file thông thường)
chmod 755 script.sh              # rwxr-xr-x (executable)
chmod 600 ~/.ssh/id_rsa          # rw------- (private key)
chmod 700 ~/.ssh                 # rwx------ (SSH dir)
chmod -R 755 /var/www/html       # recursive
chmod g+s /shared-dir            # setgid: file mới kế thừa group

chown user:group file
chown -R www-data:www-data /var/www/

# ❗ Private key chmod sai → SSH từ chối: "Permissions 0644 for key are too open"
```

I sudo tricks

```
sudo -l                          # xem quyền sudo của user hiện tại
sudo -u postgres psql            # chạy lệnh với tư cách user khác
sudo -i                          # login shell của root
sudo !!                          # chạy lại lệnh trước với sudo

# Chạy lệnh không cần password (thêm vào /etc/sudoers qua visudo):
deploy ALL=(ALL) NOPASSWD: /bin/systemctl restart myapp

# ❗ Dùng visudo để sửa sudoers, không dùng editor trực tiếp → syntax error = mất quyền sudo
```

I UFW — firewall đơn giản

```
ufw status verbose
ufw allow 22/tcp
ufw allow from 10.0.0.0/8 to any port 5432  # PostgreSQL chỉ cho internal
ufw deny 80/tcp
```

```
ufw delete allow 80/tcp
ufw enable
ufw reload

# Rate limit SSH (chống brute force):
ufw limit ssh
```

I fail2ban — chặn brute force

```
fail2ban-client status                        # xem các jail đang chạy
fail2ban-client status sshd                   # xem jail SSH
fail2ban-client set sshd banip 1.2.3.4      # ban thủ công
fail2ban-client set sshd unbanip 1.2.3.4    # unban
fail2ban-client get sshd bantime            # xem thời gian ban

# Xem log:
tail -f /var/log/fail2ban.log
```

A SELinux / AppArmor cơ bản

```
# SELinux (CentOS/RHEL):
getenforce                      # Enforcing / Permissive / Disabled
setenforce 0                    # tắt tạm (không persist)
ausearch -m avc -ts recent      # xem denial gần nhất
audit2allow -a                  # gợi ý policy từ denials

# AppArmor (Ubuntu/Debian):
aa-status                       # xem profile đang enforce
aa-complain /usr/sbin/nginx     # chuyển sang complain mode (log không block)
aa-enforce /usr/sbin/nginx      # enforce mode
journalctl | grep apparmor      # xem denials
```

---

7. Bash Scripting

B set -euo pipefail — safety defaults

```
#!/usr/bin/env bash
set -euo pipefail
# -e: exit ngay khi lệnh lỗi
# -u: lỗi nếu dùng biến chưa khai báo
# -o pipefail: pipe fail nếu bất kỳ lệnh nào trong pipe lỗi

# ! set -e có thể gây vấn đề trong một số trường hợp:
# if grep -q "pattern" file; then ... fi  → OK (exit code dùng trong điều kiện)
# grep -q "pattern" file || true          → bypass set -e khi không cần lỗi
```

I Parameter expansion

```
name="world"
echo ${name:-default}           # dùng default nếu name unset hoặc rỗng
echo ${name:=default}           # gán default nếu unset
echo ${name:?Error: name unset} # exit với message nếu unset
echo ${#name}                   # độ dài chuỗi (5)

file="/path/to/script.sh"
echo ${file##*/}                # script.sh (basename)
echo ${file%/*}                 # /path/to (dirname)
echo ${file%.sh}                # /path/to/script (xoá .sh)
echo ${file/path/src}           # /src/to/script.sh (replace first)
echo ${file//o/0}               # replace all
```

```
# Substring:
str="Hello World"
echo ${str:6}                    # World
echo ${str:0:5}                  # Hello
```

I Arrays và loops

```
# Indexed array:
servers=("web01" "web02" "db01")
echo ${servers[0]}               # web01
echo ${#servers[@]}              # 3 (số phần tử)
echo ${servers[@]}               # tất cả phần tử

for server in "${servers[@]}"; do
  echo "Deploying to $server"
done

# Associative array (Bash 4+):
declare -A config
config[host]="localhost"
config[port]="5432"
echo ${config[host]}

# ! Quote "${array[@]}" để giữ phần tử có space
```

I trap — xử lý tín hiệu và cleanup

```
#!/usr/bin/env bash
set -euo pipefail

tmpfile=$(mktemp)

cleanup() {
  echo "Cleaning up..."
  rm -f "$tmpfile"
}

trap cleanup EXIT              # chạy cleanup khi script thoát (bất kể lý do)
trap 'cleanup; exit 130' INT   # Ctrl+C
trap 'echo "Error on line $LINENO"' ERR

# Dùng trong script deploy:
trap 'systemctl rollback myapp; exit 1' ERR
```

A Process substitution

```
# Thay vì pipe (pipe tạo subshell):
diff <(ssh host1 cat /etc/hosts) <(ssh host2 cat /etc/hosts)

# Sort 2 file và diff:
diff <(sort file1) <(sort file2)

# Đọc output của lệnh như file:
while read line; do
  echo "Processing: $line"
done < <(find /var/log -name "*.log" -mtime -1)

# ! Variable thay đổi trong subshell không ảnh hưởng parent — đây là lý do dùng < <(cmd)
```

A Heredoc và herestring

```
# Heredoc:
cat > /etc/app/config.ini << EOF
host = ${DB_HOST:-localhost}
port = ${DB_PORT:-5432}
```

```
EOF

# Heredoc không expand biến (dùng khi có SQL, JSON):
cat << 'EOF'
SELECT * FROM users WHERE name = '${not_expanded}';
EOF

# Herestring:
grep "pattern" <<< "string to search in"
```

---

8. One-liners Thực Chiến

Process & System

```
# Top 10 process ngốn CPU:
ps aux --sort=-%cpu | head -11

# Đợi process kết thúc rồi chạy lệnh tiếp:
while kill -0 $PID 2>/dev/null; do sleep 1; done && echo "Done"

# Chạy lệnh mỗi khi file thay đổi:
while true; do inotifywait -e modify app.py && python app.py; done

# Xem process nào đang dùng nhiều file nhất:
lsof | awk '{print $2}' | sort | uniq -c | sort -rn | head
```

Disk & Filesystem

```
# Tìm 20 file lớn nhất trên hệ thống:
find / -type f -printf '%s %p\n' 2>/dev/null | sort -rn | head -20

# Dung lượng thư mục, sort:
du -sh */ | sort -rh | head -20

# Xoá file log cũ hơn 30 ngày:
find /var/log/myapp -name "*.log" -mtime +30 -exec rm {} \;

# Đếm số file trong thư mục (đệ quy):
find . -type f | wc -l
```

Networking

```
# Đếm kết nối theo trạng thái:
ss -s

# Top IP kết nối nhiều nhất:
ss -tn | awk 'NR>1 {print $5}' | cut -d: -f1 | sort | uniq -c | sort -rn | head

# Test xem port có mở không (không cần telnet):
(echo >/dev/tcp/host/80) 2>/dev/null && echo "open" || echo "closed"

# Download file song song:
cat urls.txt | xargs -P 5 -n1 wget -q
```

Text & Log

```
# Theo dõi nhiều log cùng lúc:
tail -f /var/log/nginx/access.log /var/log/nginx/error.log

# Đếm request theo HTTP status trong access.log:
awk '{print $9}' /var/log/nginx/access.log | sort | uniq -c | sort -rn
```

```
# Tìm 10 URL được request nhiều nhất:
awk '{print $7}' /var/log/nginx/access.log | sort | uniq -c | sort -rn | head -10

# Extract tất cả IP từ log:
grep -oE '\b([0-9]{1,3}\.){3}[0-9]{1,3}\b' access.log | sort -u

# Convert Unix timestamp trong log:
awk '{print strftime("%Y-%m-%d %H:%M:%S", $1), $0}' timestamps.log

# Theo dõi log và alert khi có "error":
tail -f app.log | grep --line-buffered -i error | while read line; do
  echo "ALERT: $line" | mail -s "App Error" admin@example.com
done
```

SSH & Remote

```
# Copy SSH key lên server:
ssh-copy-id -i ~/.ssh/id_ed25519.pub user@host

# SSH tunnel (forward port local 5432 → remote DB):
ssh -L 5432:db-server:5432 jump-host -N -f

# Chạy lệnh trên nhiều server:
for h in web01 web02 web03; do ssh $h "sudo systemctl status nginx"; done

# Sync thư mục và xem progress:
rsync -avz --progress /local/path/ user@host:/remote/path/

# ❗ ssh -N -f: -N không execute command, -f background sau auth
```

Miscellaneous

```
# Chạy lệnh và đo thời gian:
time ./long-script.sh

# Thực thi lệnh sau khoảng thời gian:
sleep 3600 && systemctl restart myapp    # restart sau 1 giờ

# Xem history với timestamp:
HISTTIMEFORMAT="%F %T " history | tail -20

# Backup nhanh file trước khi sửa:
cp config.yml{,.bak.$(date +%Y%m%d)}    # tạo config.yml.bak.20240101

# Kiểm tra script syntax không chạy:
bash -n script.sh
```

Cập nhật: 2026-05-27 | Nền tảng: Ubuntu/Debian, tương thích phần lớn với RHEL/CentOS

02

Git Tricks — Từ Cơ Bản Đến Nâng Cao

> Giải thích bằng tiếng Việt, lệnh bằng tiếng Anh. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

1. History & Search

B git log — định dạng đẹp hơn

```
git log --oneline --graph --decorate --all
# Xem toàn bộ lịch sử dạng graph

git log --pretty=format:"%C(yellow)%h%Creset %C(cyan)%ad%Creset %s %C(green)[%an]%Creset" --
date=short
# hash | ngày | message | author

git log -10                          # 10 commit gần nhất
git log --since="2 weeks ago"
git log --author="Hieu"
git log -- path/to/file              # lịch sử của file cụ thể
git log -p -- path/to/file           # lịch sử + diff của file

# Alias hữu ích (thêm vào ~/.gitconfig):
git config --global alias.lg "log --oneline --graph --decorate --all"
# Sau đó dùng: git lg
```

I git log -S và -G — tìm theo nội dung

```
# -S: tìm commit thêm/xoá string (pickaxe search)
git log -S "password" --all          # commit nào đã chứa "password"
git log -S "database_url" -p         # + xem diff

# -G: tìm commit có diff match regex
git log -G "def process_payment" -p  # commit thay đổi hàm này
git log -G "TODO|FIXME" --oneline    # commit có TODO/FIXME trong diff

# Tìm trong tất cả branch:
git log -S "secret_key" --all --oneline
```

I git blame — ai sửa dòng này

```
git blame -L 10,20 file.py           # blame dòng 10-20
git blame -w file.py                 # bỏ qua whitespace changes
git blame --follow -C file.py        # theo dõi cả khi file bị rename/copy
git blame -e file.py                 # hiển thị email thay vì username

# ! Blame chỉ hiển thị commit cuối chạm vào dòng đó
# Nếu cần lịch sử đầy đủ: git log -p -S "dòng cần tìm" -- file.py
```

I git shortlog — thống kê contributor

```
git shortlog -sn                    # số commit theo author, sort
git shortlog -sn --all              # bao gồm tất cả branch
git shortlog -sn --since="1 month ago"
git shortlog -e -sn                 # kèm email

# Đếm dòng code theo author (cần git log + awk):
git log --numstat --pretty="%ae" | awk '/^[^@]*@[^@]*$/ {author=$0} /^[0-9]/ {added[author]+=$1;
removed[author]+=$2} END {for (a in added) print added[a]-removed[a], a}' | sort -rn
```

A git grep — tìm trong codebase (nhanh hơn grep thường)

```
git grep "TODO"                     # tìm trong working tree
git grep "deprecated" HEAD~10       # tìm trong commit cụ thể
git grep -n "function handleAuth"   # kèm số dòng
git grep -l "import redis"          # chỉ in tên file
git grep -E "class (User|Admin)"    # regex

# ! git grep chỉ tìm trong tracked files và bỏ qua .gitignore — nhanh hơn grep -r nhiều
```

---

2. Branching

B switch vs checkout — lệnh mới hơn

```
# git switch (Git 2.23+) — rõ ràng hơn checkout
git switch main                     # chuyển branch
git switch -c feature/login         # tạo và chuyển sang branch mới
git switch -                        # về branch trước đó

# git checkout — vẫn dùng nhưng đa nghĩa hơn:
git checkout main                   # chuyển branch
git checkout -b feature/login       # tạo branch
git checkout -- file.py             # discard changes (hay nhầm với switch)
git restore file.py                 # lệnh mới, rõ ràng hơn cho discard changes

# ! git checkout -- file.py và git switch đều checkout nhưng khác mục đích
# Dùng git switch cho branch, git restore cho file để tránh nhầm
```

I git worktree — làm việc nhiều branch cùng lúc

```
# Checkout branch khác vào thư mục riêng, không cần stash/commit dở
git worktree add ../feature-auth feature/auth
git worktree add ../hotfix-prod -b hotfix/prod-crash origin/main

# Liệt kê worktrees:
git worktree list

# Xoá worktree:
git worktree remove ../feature-auth
git worktree prune                  # dọn worktree không còn tồn tại trên disk

# ! Không thể checkout cùng một branch vào 2 worktree khác nhau
# Mỗi worktree là một working directory độc lập, share cùng .git
```

I Dọn dẹp branch

```
# Xoá branch local đã merge:
git branch --merged main | grep -v "^\*\|main\|develop" | xargs git branch -d

# Xoá branch remote đã delete:
```

```
git fetch --prune                    # tự động xoá remote-tracking branch không còn

# Xem branch chưa merge vào main:
git branch --no-merged main

# Xoá tất cả local branch đã merged (nguy hiểm, xác nhận kỹ):
git branch --merged | grep -v "\*\|main\|master\|develop" | xargs -r git branch -d

# ❗ git branch -d sẽ lỗi nếu branch chưa merge. Dùng -D để force delete
```

A Tracking branch và upstream

```
git branch -u origin/main main       # set upstream cho local branch
git branch -vv                       # xem upstream của tất cả branch
git push -u origin feature/login     # push và set upstream cùng lúc

# Xem remote branch:
git branch -r
git branch -a                        # local + remote

# Fetch về không merge:
git fetch origin
git diff main origin/main            # xem thay đổi trước khi merge/pull
```

---

3. Rewriting History

I git rebase -i — sửa lịch sử interactive

```
git rebase -i HEAD~5                 # sửa 5 commit gần nhất
git rebase -i origin/main           # sửa commit chưa push lên main

# Trong editor xuất hiện các lệnh:
# pick   → giữ nguyên
# reword → sửa message
# edit   → dừng lại để sửa code
# squash → gộp vào commit trước, giữ message
# fixup  → gộp vào commit trước, bỏ message
# drop   → xoá commit

# ❗ KHÔNG rebase branch đã push public — sẽ tạo diverge với người khác
# Chỉ rebase local branch hoặc branch chỉ mình bạn dùng
```

I fixup và autosquash — workflow sạch

```
# Workflow: tạo fixup commit, sau đó squash tự động
git add file.py
git commit --fixup HEAD~2            # tạo commit "fixup! <message của HEAD~2>"

# Squash tất cả fixup commits tự động:
git rebase -i --autosquash HEAD~5
# Git tự sắp xếp fixup! commits đúng vị trí và dùng fixup action

# Setup autosquash mặc định:
git config --global rebase.autosquash true
```

I git cherry-pick — lấy commit cụ thể

```
git cherry-pick abc1234              # áp dụng commit vào branch hiện tại
git cherry-pick abc1234..def5678     # range commits (không bao gồm abc1234)
git cherry-pick abc1234^..def5678    # bao gồm cả abc1234
git cherry-pick -n abc1234           # áp dụng thay đổi nhưng không commit
```

```
git cherry-pick --continue          # sau khi resolve conflict
git cherry-pick --abort

# ⚠ cherry-pick tạo commit MỚI với hash khác, dù code giống hệt
# Nếu sau này merge branch gốc, có thể bị duplicate changes
```

A git filter-repo — rewrite history (thay filter-branch)

```
# Cài: pip install git-filter-repo

# Xoá file nhạy cảm khỏi toàn bộ lịch sử:
git filter-repo --path secrets.env --invert-paths

# Đổi tên file trong toàn bộ lịch sử:
git filter-repo --path-rename old-name.py:new-name.py

# Xoá tất cả file lớn hơn 10MB:
git filter-repo --strip-blobs-bigger-than 10M

# Thay email trong toàn bộ lịch sử:
git filter-repo --email-callback 'return email.replace(b"old@email.com", b"new@email.com")'

# ⚠ filter-repo xóa remote tracking sau khi chạy
# Cần git remote add origin <url> lại rồi force push: git push --force-with-lease --all
# ⚠ Tất cả collaborator cần clone lại sau khi rewrite history
```

A git rebase — rebase onto

```
# Di chuyển branch sang base mới:
git rebase --onto main feature-base feature-top
# Lấy commits từ feature-base..feature-top và áp dụng lên main

# Ví dụ: feature/login dựa trên feature/auth, muốn đặt trực tiếp lên main:
git rebase --onto main feature/auth feature/login
```

---

4. Recovery

B git reflog — thùng rác của Git

```
git reflog                          # mọi thay đổi HEAD trong 90 ngày
git reflog show feature/login       # reflog của branch cụ thể

# Recover branch đã xoá:
git reflog                          # tìm hash của commit cuối trên branch đó
git checkout -b feature/login abc1234  # tạo lại branch từ hash

# Recover sau git reset --hard:
git reflog                          # tìm commit trước khi reset
git reset --hard HEAD@{3}           # quay về 3 bước trước
```

I Recover file đã xoá

```
# File xoá chưa commit:
git restore deleted-file.py         # lấy lại từ index/HEAD

# File xoá đã commit:
git log --all --full-history -- deleted-file.py   # tìm commit xoá file
git show HASH:deleted-file.py > restored.py        # lấy lại nội dung

# Lấy file từ commit cụ thể:
git checkout HASH -- path/to/file.py  # restore file từ commit đó vào working tree
```

I Recover stashed changes

```
git stash list                          # liệt kê tất cả stash
git stash show -p stash@{2}             # xem nội dung stash
git stash pop                           # áp dụng stash mới nhất + xoá khỏi list
git stash apply stash@{2}               # áp dụng stash cụ thể + giữ trong list
git stash drop stash@{2}                # xoá stash

# Stash đã drop vẫn có thể recover:
git fsck --unreachable | grep commit | awk '{print $3}' | xargs git show --stat
# Tìm stash hash rồi: git stash apply HASH
```

A git fsck — kiểm tra và recover object

```
git fsck --unreachable                  # tìm object không được reference
git fsck --lost-found                   # lưu unreachable object vào .git/lost-found/

# Recover commit orphan (không có branch trỏ vào):
git fsck --unreachable 2>/dev/null | grep "^unreachable commit" | awk '{print $3}' | \
  xargs -I{} git log -1 --pretty="%H %s" {}

# ! Object bị garbage collect sau 90 ngày (mặc định gc.reflogExpire)
# Sau 90 ngày: không thể recover qua reflog, có thể qua fsck nếu chưa gc
```

---

5. Bisect

I git bisect — tìm commit gây bug

```
git bisect start
git bisect bad                          # commit hiện tại có bug
git bisect good v1.0.0                  # tag/commit đã biết là tốt
# Git tự checkout commit ở giữa, bạn test rồi đánh dấu:
git bisect good                         # hoặc:
git bisect bad
# Lặp cho đến khi tìm được commit gây bug
git bisect reset                        # kết thúc, về HEAD ban đầu
```

A git bisect run — tự động với script

```
# Viết script test trả về 0 = good, non-zero = bad
cat > test-bug.sh << 'EOF'
#!/bin/bash
make build && ./run-tests.sh --test login
EOF
chmod +x test-bug.sh

git bisect start
git bisect bad HEAD
git bisect good v1.0.0
git bisect run ./test-bug.sh
# Git tự động tìm commit gây bug mà không cần can thiệp thủ công
git bisect reset

# ! Script phải trả về exit code 125 nếu không thể test commit đó (skip)
# Ví dụ: commit không build được → exit 125 để git bisect skip
```

---

6. Performance

B Shallow clone — clone nhanh hơn

```
git clone --depth=1 https://github.com/org/repo.git
# Chỉ clone 1 commit gần nhất — rất nhanh, phù hợp CI/CD

git clone --depth=10 --branch main https://github.com/org/repo.git
# 10 commit gần nhất, branch cụ thể

# Unshallow khi cần lịch sử đầy đủ:
git fetch --unshallow

# ! Shallow clone không thể git blame đầy đủ hoặc bisect
```

I Sparse checkout — chỉ checkout một phần repo

```
# Monorepo lớn, chỉ cần một subdirectory:
git clone --filter=blob:none --sparse https://github.com/org/monorepo.git
cd monorepo
git sparse-checkout set apps/backend docs/api
# Chỉ checkout thư mục apps/backend và docs/api

git sparse-checkout list             # xem patterns hiện tại
git sparse-checkout add apps/frontend  # thêm thư mục
git sparse-checkout disable          # tắt sparse checkout
```

I Partial clone — không download blob

```
git clone --filter=blob:none https://github.com/org/repo.git
# Clone không tải blob (file content), chỉ tree/commit
# Blob tải về khi cần (lazy fetch)

git clone --filter=tree:0 https://github.com/org/repo.git
# Không tải tree của commit cũ — chỉ commit metadata

# Phù hợp với: repo có nhiều file lớn, CI chỉ cần build 1 phần
```

A git maintenance — tự động tối ưu

```
git maintenance start                # bật maintenance tự động (cron)
git maintenance run                  # chạy ngay
git maintenance run --task gc        # chỉ garbage collection
git maintenance run --task commit-graph  # build commit-graph để log nhanh hơn
git maintenance run --task prefetch  # prefetch remote object

# Xem các task:
git maintenance run --task=?         # list tasks

# ! git maintenance start thêm cron job vào user's crontab
# Kiểm tra: crontab -l | grep git
```

A commit-graph — tăng tốc git log

```
git commit-graph write --reachable
# Tạo file .git/objects/info/commit-graph
# git log, git status, git branch nhanh hơn nhiều trên repo lớn

# Tự động update (đã có nếu dùng git maintenance):
git config core.commitGraph true
git config gc.writeCommitGraph true
```

7. Hooks

pre-commit — kiểm tra trước khi commit

```
# .git/hooks/pre-commit (chmod +x)
#!/usr/bin/env bash
set -euo pipefail

# Chạy linter:
echo "Running linters..."
flake8 . --count --select=E9,F63,F7,F82 --show-source

# Kiểm tra secrets (greppin đơn giản):
if git diff --cached | grep -E "(password|secret|api_key)\s*=\s*['\"][^'\"]+['\"]" -i; then
  echo "ERROR: Possible secret detected in diff!"
  exit 1
fi

echo "Pre-commit checks passed."
```

commit-msg — enforce format

```
# .git/hooks/commit-msg
#!/usr/bin/env bash
commit_msg=$(cat "$1")
pattern="^(feat|fix|docs|style|refactor|test|chore|ci|perf)(\(.+\))?: .{1,72}"

if ! echo "$commit_msg" | grep -qE "$pattern"; then
  echo "ERROR: Commit message không đúng format conventional commits."
  echo "Ví dụ: feat(auth): add JWT refresh token"
  exit 1
fi
```

pre-push — chạy tests trước khi push

```
# .git/hooks/pre-push
#!/usr/bin/env bash
set -euo pipefail

protected_branch="main"
current_branch=$(git rev-parse --abbrev-ref HEAD)

if [ "$current_branch" = "$protected_branch" ]; then
  echo "Running tests before push to $protected_branch..."
  npm test || { echo "Tests failed! Push aborted."; exit 1; }
fi
```

Chia sẻ hooks với team qua pre-commit framework

```
# Dùng pre-commit (pip install pre-commit) thay vì viết tay
# .pre-commit-config.yaml:
cat > .pre-commit-config.yaml << 'EOF'
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: detect-private-key
  - repo: https://github.com/psf/black
    rev: 23.12.1
    hooks:
      - id: black
```

```
EOF

pre-commit install                    # cài hook vào .git/hooks/
pre-commit run --all-files            # chạy thủ công

# ❗ Hooks trong .git/hooks/ không được commit vào repo
# Dùng pre-commit framework để share hooks qua .pre-commit-config.yaml
```

---

8. Config Tricks

I .gitattributes — kiểm soát merge và diff

```
# .gitattributes
# Line endings:
* text=auto
*.sh text eol=lf
*.bat text eol=crlf
*.png binary

# Diff driver cho file cụ thể:
*.ipynb diff=json
*.pdf diff=pdf

# Merge strategy:
package-lock.json merge=ours        # luôn giữ version của mình khi merge

# Export ignore (không include khi git archive):
.gitignore export-ignore
docs/ export-ignore
```

I Merge drivers tuỳ chỉnh

```
# Tự động resolve conflict cho package-lock.json bằng npm:
git config merge.npm-merge-driver.driver \
  "npx npm-merge-driver merge %O %A %B %P"

# Trong .gitattributes:
# package-lock.json merge=npm-merge-driver
```

I Diff algorithms

```
# Algorithm mặc định (Myers) đôi khi tạo diff khó đọc
git diff --diff-algorithm=histogram  # thường dễ đọc hơn cho code
git diff --diff-algorithm=patience   # tốt cho refactor

# Set mặc định:
git config --global diff.algorithm histogram

# Diff word-by-word thay vì line-by-line:
git diff --word-diff
git diff --color-words
```

A rerere — tự nhớ cách resolve conflict

```
git config --global rerere.enabled true
# rerere = Reuse Recorded Resolution
# Git ghi nhớ cách bạn resolve conflict
# Lần sau gặp conflict giống hệt → tự động resolve

git rerere status                    # xem conflict đã ghi nhớ
git rerere diff                      # xem resolution đã lưu
```

```
git rerere forget path/to/file       # xoá resolution đã lưu

# Hữu ích khi: rebase dài, merge nhiều lần cùng conflict
```

Conditional config — config theo thư mục

```
# ~/.gitconfig
[includeIf "gitdir:~/work/"]
  path = ~/.gitconfig-work
[includeIf "gitdir:~/personal/"]
  path = ~/.gitconfig-personal

# ~/.gitconfig-work:
[user]
  email = hieu@company.com
  signingkey = WORK_GPG_KEY

# ~/.gitconfig-personal:
[user]
  email = personal@gmail.com
```

Useful global configs

```
git config --global pull.rebase true          # pull = fetch + rebase, không tạo merge commit
git config --global push.default current      # push branch hiện tại lên remote cùng tên
git config --global fetch.prune true          # tự prune khi fetch
git config --global init.defaultBranch main   # branch mặc định là main
git config --global core.autocrlf input       # LF trên Linux/Mac
git config --global rebase.autostash true     # tự stash khi rebase có uncommitted changes

# Editor:
git config --global core.editor "vim"
git config --global core.editor "code --wait"  # VS Code
```

---

9. Collaboration

format-patch — tạo patch để gửi email

```
git format-patch origin/main         # tạo .patch file cho mỗi commit chưa push
git format-patch -3                  # 3 commit gần nhất
git format-patch HEAD~3..HEAD --stdout > changes.patch  # 1 file

# Áp dụng patch:
git am changes.patch                 # apply và tạo commit
git apply changes.patch              # chỉ apply, không commit

# ⚠ format-patch tạo commit đúng author/timestamp
# git apply chỉ apply changes, không giữ author
```

git bundle — repo offline

```
# Đóng gói repo để chuyển qua USB/không có mạng:
git bundle create repo.bundle --all

# Giải nén:
git clone repo.bundle local-repo
# Hoặc fetch vào repo hiện có:
git remote add bundle /path/to/repo.bundle
git fetch bundle
```

```
# Tạo bundle chỉ những commit chưa có ở đích:
git bundle create updates.bundle origin/main..HEAD
```

A Patch workflow với send-email

```
# Cấu hình SMTP (thường dùng cho kernel/open source contribution):
git config sendemail.smtpserver smtp.gmail.com
git config sendemail.smtpserverport 587
git config sendemail.smtpencryption tls
git config sendemail.smtpuser your@email.com

# Tạo và gửi patch:
git format-patch -1 HEAD
git send-email --to=maintainer@project.org *.patch
git send-email --cover-letter --annotate -3  # gửi 3 commit với cover letter
```

I git shortstat và diff stats

```
git diff --stat main                # tổng số file/dòng thay đổi
git diff --numstat main             # dạng số (dễ parse)
git diff --dirstat main             # % thay đổi theo thư mục

# So sánh branch:
git log --left-right --count main...feature/login
# 5 < 3 → main có 5 commit feature không có, feature/login có 3 commit main không có
```

Quick Reference: Tình Huống Thực Tế

Lỡ commit nhạy cảm (password, token)

```
# 1. Xoá ngay trên remote bằng filter-repo:
git filter-repo --path secrets.env --invert-paths
git push --force-with-lease --all
git push --force-with-lease --tags

# 2. Revoke token/password ngay lập tức — git history vẫn có trong cache các clone cũ
# 3. Thông báo team clone lại
```

Merge conflict khi rebase

```
git rebase origin/main
# Conflict xảy ra...
git status                          # xem file conflict
# Sửa file, rồi:
git add resolved-file.py
git rebase --continue
# Hoặc bỏ qua commit này:
git rebase --skip
# Hoặc abort hoàn toàn:
git rebase --abort
```

Undo commit đã push

```
# Cách an toàn (không rewrite history, phù hợp branch public):
git revert HASH                     # tạo commit mới đảo ngược
git revert HEAD~3..HEAD             # revert 3 commit gần nhất
git push

# Cách mạnh (rewrite history, chỉ dùng cho branch riêng):
git reset --hard HEAD~1
```

```
git push --force-with-lease          # an toàn hơn --force (kiểm tra remote không thay đổi)

# ❗ --force-with-lease thay vì --force để tránh ghi đè commit của người khác
```

Lấy file từ branch khác mà không merge

```
git checkout feature/payment -- src/payment/handler.py
# Lấy file vào working tree và index, không tạo commit
git restore --source=feature/payment -- src/payment/handler.py  # lệnh mới hơn
```

Stash chọn lọc

```
git stash push -p                    # interactive, chọn hunk để stash
git stash push -m "WIP: login form" src/auth/  # stash chỉ thư mục cụ thể
git stash push --include-untracked   # bao gồm file chưa track
```

Cập nhật: 2026-05-27 | Git phiên bản 2.40+

03

Docker Tricks: Từ Beginner đến Advanced

> Cập nhật: 2026-05-27 | Phong cách: thực chiến, không lý thuyết thừa

---

1. Image Building

B Multi-stage build — giảm image size mạnh

Multi-stage tách builder và runtime ra hai stage riêng. Stage cuối chỉ copy artifact, không mang theo compiler/dependencies thừa.

```
# Stage 1: build
FROM golang:1.22-alpine AS builder
WORKDIR /app
COPY . .
RUN go build -o server .

# Stage 2: runtime (chỉ ~15MB)
FROM scratch
COPY --from=builder /app/server /server
ENTRYPOINT ["/server"]
```

```
docker build -t myapp:latest .
docker images myapp
# myapp   latest   abc123   5 seconds ago   8.5MB
```

! Nếu dùng `scratch`, binary phải static-linked. Với Go: `CGO_ENABLED=0 go build`.

B .dockerignore — tránh copy rác vào build context

Không có `.dockerignore` thì toàn bộ `node_modules`, `.git`, logs sẽ được gửi lên daemon mỗi lần build.

```
# .dockerignore
node_modules
.git
*.log
dist
.env
.DS_Store
coverage/
```

```
# Kiểm tra build context size trước và sau
docker build --no-cache . 2>&1 | head -5
# Sending build context to Docker daemon  1.234MB  ← mục tiêu
```

Layer caching — thứ tự instruction quan trọng

Docker cache theo layer từ trên xuống. Đặt những gì ít thay đổi lên trên, source code xuống dưới.

```
# Sai: copy toàn bộ trước, npm install sau → mỗi lần đổi code là reinstall deps
COPY . .
RUN npm install

# Đúng: chỉ copy package.json trước
COPY package*.json ./
RUN npm install
COPY . .
```

BuildKit secrets — không để secret trong layer

BuildKit cho phép mount secret tại build time mà không bake vào image.

```
# Enable BuildKit
export DOCKER_BUILDKIT=1

# Dockerfile
RUN --mount=type=secret,id=npmrc,dst=/root/.npmrc \
    npm install
```

```
docker build --secret id=npmrc,src=$HOME/.npmrc -t myapp .
```

Secret không xuất hiện trong `docker history` hay `docker inspect`.

Build args vs ENV — biết cái nào dùng lúc nào

`ARG` chỉ tồn tại trong quá trình build. `ENV` tồn tại cả trong container runtime.

```
ARG APP_VERSION=1.0.0          # chỉ dùng khi build
ENV NODE_ENV=production         # tồn tại khi container chạy

RUN echo "Building version $APP_VERSION"
```

```
docker build --build-arg APP_VERSION=2.0.0 -t myapp .
```

`ARG` trước `FROM` dùng để parameterize base image. `ARG` sau `FROM` thì scoped trong stage đó.

Heredoc trong Dockerfile (BuildKit)

Viết script inline không cần file riêng.

```
RUN <<EOF
set -e
apt-get update
apt-get install -y curl
rm -rf /var/lib/apt/lists/*
EOF
```

---

2. Layer Optimization

B Combine RUN commands — giảm số layer

Mỗi `RUN` tạo một layer. Combine để giảm size.

```
# Sai: 3 layer riêng
RUN apt-get update
RUN apt-get install -y curl wget
RUN rm -rf /var/lib/apt/lists/*

# Đúng: 1 layer, sạch hơn
RUN apt-get update && \
    apt-get install -y --no-install-recommends curl wget && \
    rm -rf /var/lib/apt/lists/*
```

B Dùng Alpine/Distroless thay full OS

```
# Node.js: ~900MB → ~120MB
FROM node:20-alpine

# Distroless (Google): không có shell, attack surface tối thiểu
FROM gcr.io/distroless/nodejs20-debian12
```

```
docker images | grep -E "node:20$|node:20-alpine|distroless"
# node     20           1.1GB
# node     20-alpine    133MB
# distroless/nodejs20   115MB
```

I Dùng dive để phân tích layer

```
# Cài dive
brew install dive  # macOS
# hoặc
docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock wagoodman/dive myapp:latest
```

Dive hiển thị từng layer, file nào bị thêm/xóa, wasted space ở đâu.

A Scratch image cho static binary

```
FROM scratch
COPY --from=builder /etc/ssl/certs/ca-certificates.crt /etc/ssl/certs/
COPY --from=builder /app/server /server
EXPOSE 8080
ENTRYPOINT ["/server"]
```

Image cuối chỉ có binary + CA certs, không có gì khác. Size thường < 20MB.

3. Runtime

B Resource limits — tránh container ngốn hết tài nguyên host

```
docker run -d \
  --memory="512m" \
```

```
  --memory-swap="512m" \          # disable swap
  --cpus="1.5" \
  nginx
```

```
docker stats --no-stream
# CONTAINER   CPU %   MEM USAGE / LIMIT   MEM %
# nginx_1     0.2%    45MB / 512MB        8.8%
```

I Read-only filesystem + tmpfs

Chạy container với filesystem read-only, chỉ cho phép ghi vào tmpfs (RAM).

```
docker run -d \
  --read-only \
  --tmpfs /tmp:rw,noexec,nosuid,size=100m \
  --tmpfs /var/run:rw,noexec,nosuid \
  nginx
```

! App phải không ghi ra ngoài `/tmp` và `/var/run`. Kiểm tra kỹ trước khi dùng production.

I PID namespace isolation

Mặc định container share PID namespace với host (không). Dùng `--pid=host` khi cần debug process host từ trong container (cẩn thận bảo mật).

```
# Debug: xem process host từ container
docker run --rm --pid=host alpine ps aux | head -20

# Production: tuyệt đối không dùng --pid=host
```

A Seccomp profiles — lọc syscall

```
# Xem syscall profile mặc định Docker dùng
docker info --format '{{.SecurityOptions}}'
# name=seccomp,profile=builtin

# Custom profile
docker run --security-opt seccomp=/path/to/custom.json nginx
```

! Sai seccomp profile có thể làm app crash mà không có error rõ ràng. Test kỹ.

---

4. Networking

B Bridge vs Host vs None

```
# Bridge (mặc định): container có private IP, NAT ra ngoài
docker run -d --network bridge nginx

# Host: dùng thẳng network stack của host, không NAT overhead
docker run -d --network host nginx

# None: không có network
docker run -d --network none alpine sleep infinity
```

! `--network host` không hoạt động trên macOS/Windows Docker Desktop vì Docker chạy trong VM.

B Custom network — container giao tiếp qua tên

```
docker network create myapp-net

docker run -d --name db --network myapp-net postgres:16
docker run -d --name api --network myapp-net myapi

# Từ container api, gọi db bằng hostname "db"
# Không cần biết IP
```

I Port publishing internals

`-p 8080:80` không phải magic. Docker tạo iptables rule chuyển traffic từ host:8080 → container:80.

```
# Xem iptables rule Docker tạo
sudo iptables -t nat -L DOCKER -n --line-numbers

# Bind cụ thể IP (tránh expose ra internet không mong muốn)
docker run -p 127.0.0.1:8080:80 nginx
```

! `-p 0.0.0.0:8080:80` (mặc định) expose ra tất cả interfaces, kể cả public IP.

I DNS resolution trong container

```
# Xem DNS container đang dùng
docker run --rm alpine cat /etc/resolv.conf
# nameserver 127.0.0.11  ← Docker embedded DNS

# Debug DNS lookup
docker run --rm --network myapp-net nicolaka/netshoot \
    dig db.myapp-net
```

5. Volumes & Storage

B Named volumes vs Bind mounts

```
# Named volume: Docker quản lý, persist khi container xóa
docker run -d -v postgres_data:/var/lib/postgresql/data postgres:16

# Bind mount: sync với thư mục host (dev workflow)
docker run -d -v $(pwd):/app -w /app node:20 npm run dev
```

I Volume backup pattern

```
# Backup named volume ra tar
docker run --rm \
  -v postgres_data:/source:ro \
  -v $(pwd)/backups:/backup \
  alpine tar czf /backup/postgres_$(date +%Y%m%d).tar.gz -C /source .

# Restore
docker run --rm \
  -v postgres_data:/target \
```

```
  -v $(pwd)/backups:/backup \
  alpine tar xzf /backup/postgres_20260527.tar.gz -C /target
```

A Volume drivers — Distributed storage

```
# Dùng NFS volume driver
docker volume create \
  --driver local \
  --opt type=nfs \
  --opt o=addr=192.168.1.100,rw \
  --opt device=:/exports/data \
  nfs_volume

docker run -d -v nfs_volume:/data myapp
```

6. Debugging

B Docker exec tricks

```
# Vào container đang chạy
docker exec -it container_name bash

# Chạy lệnh không cần shell tương tác
docker exec container_name env
docker exec container_name cat /etc/hosts

# Chạy với user khác
docker exec -u root container_name bash
```

B Docker cp — copy file ra/vào container

```
# Copy file từ container ra host
docker cp container_name:/app/logs/error.log ./error.log

# Copy file từ host vào container
docker cp ./config.yaml container_name:/app/config.yaml
```

I nsenter — debug ở kernel level

Khi container không có shell (distroless), dùng `nsenter` từ host.

```
# Lấy PID của container trên host
PID=$(docker inspect --format '{{.State.Pid}}' container_name)

# Vào network namespace của container
nsenter -t $PID -n ip addr

# Vào mount namespace
nsenter -t $PID -m ls /app
```

I Docker diff — xem file nào thay đổi

```
docker diff container_name
# A /tmp/newfile          ← Added
```

```
# C /etc/hosts          ← Changed
# D /var/cache/deleted  ← Deleted
```

I Inspect layers với docker history

```
docker history myapp:latest --no-trunc
# IMAGE      CREATED    CREATED BY                SIZE
# abc123     1 min ago  CMD ["node", "server.js"]  0B
# def456     1 min ago  COPY . /app               2.3MB
```

A Ephemeral debug container (Docker 23+)

```
# Attach debug sidecar vào container đang chạy
docker debug container_name --image nicolaka/netshoot
```

7. Docker Compose

B Override files — tách config dev/prod

```
# docker-compose.yml: base config
# docker-compose.override.yml: dev overrides (auto-loaded)
# docker-compose.prod.yml: prod overrides

# Dev (tự động merge override.yml)
docker compose up

# Production
docker compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

B Profiles — chỉ start service cần thiết

```
services:
  api:
    image: myapi
    # không có profile → luôn start

  redis:
    image: redis
    profiles: [cache]

  mailhog:
    image: mailhog/mailhog
    profiles: [dev]
```

```
docker compose --profile dev up        # start api + mailhog
docker compose --profile cache up      # start api + redis
docker compose --profile dev --profile cache up  # tất cả
```

I depends_on với healthcheck

`depends_on` mặc định chỉ đợi container start, không đợi service ready.

```
services:
  db:
```

```
    image: postgres:16
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

  api:
    image: myapi
    depends_on:
      db:
        condition: service_healthy
```

I env_file — quản lý environment variables

```
services:
  api:
    env_file:
      - .env              # shared vars
      - .env.local        # local overrides (không commit)
    environment:
      NODE_ENV: production  # override cụ thể
```

! Thứ tự: `env_file` trước, `environment` sau, `environment` thắng.

A Compose extends — tái sử dụng config

```
# base.yml
services:
  base-api:
    image: myapi
    restart: always
    logging:
      driver: json-file
      options: {max-size: "10m", max-file: "3"}

# docker-compose.yml
services:
  api-v1:
    extends:
      file: base.yml
      service: base-api
    ports: ["8080:8080"]

  api-v2:
    extends:
      file: base.yml
      service: base-api
    ports: ["8081:8080"]
```

---

8. Security

B Chạy với non-root user

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

```
docker run --user 1001:1001 myapp
```

⚠ Kiểm tra ownership của files app cần đọc/ghi trước khi đổi user.

I Rootless Docker

Chạy Docker daemon không cần root. Cô lập hoàn toàn với host.

```
# Cài rootless Docker
dockerd-rootless-setuptool.sh install

# Dùng
export DOCKER_HOST=unix:///run/user/$(id -u)/docker.sock
docker run hello-world
```

I Scan image với Trivy/Docker Scout

```
# Trivy (miễn phí, open source)
trivy image myapp:latest

# Docker Scout (tích hợp sẵn Docker Desktop)
docker scout cves myapp:latest
docker scout recommendations myapp:latest
```

A User namespaces — remap UID trong container

Root trong container → UID thường trên host.

```
# /etc/docker/daemon.json
{
  "userns-remap": "default"
}

# Root (UID 0) trong container = UID 165536 trên host
```

⚠ Không tương thích với `--network host` và một số volume configurations.

---

9. Performance

B Log rotation — tránh disk đầy

```
# /etc/docker/daemon.json
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

Hoặc per-container:

```
docker run --log-opt max-size=10m --log-opt max-file=3 nginx
```

B Prune định kỳ — dọn rác

```
# Xóa tất cả resource không dùng
docker system prune -a --volumes

# Chỉ xóa image
docker image prune -a

# Xóa build cache cũ hơn 48 giờ
docker builder prune --filter until=48h
```

I Registry mirror/cache — build nhanh hơn trong CI

```
// /etc/docker/daemon.json
{
  "registry-mirrors": ["https://mirror.gcr.io"],
  "insecure-registries": ["registry.internal:5000"]
}
```

A Overlay2 tuning

```
# Kiểm tra storage driver
docker info | grep "Storage Driver"
# Storage Driver: overlay2

# Kiểm tra overlay2 options
cat /etc/docker/daemon.json
# {"storage-driver": "overlay2", "storage-opts": ["overlay2.override_kernel_check=true"]}
```

---

10. Production Patterns

B Healthcheck trong Dockerfile

```
HEALTHCHECK --interval=30s --timeout=10s --start-period=10s --retries=3 \
    CMD curl -f http://localhost:8080/health || exit 1
```

```
docker ps --format "table {{.Names}}\t{{.Status}}"
# myapp   Up 2 hours (healthy)
```

I Graceful shutdown — xử lý SIGTERM

Mặc định Docker gửi SIGTERM, đợi 10s rồi SIGKILL.

```
# Dùng exec form (không qua shell) để signal đến thẳng process
ENTRYPOINT ["node", "server.js"]
# Tránh: ENTRYPOINT ["sh", "-c", "node server.js"]  ← shell không forward signal
```

```
# Tăng timeout nếu app cần cleanup lâu
docker stop --time 30 container_name
```

Init process — tránh zombie processes

```
# Tini init process
docker run --init myapp

# Trong Dockerfile
FROM node:20-alpine
RUN apk add --no-cache tini
ENTRYPOINT ["/sbin/tini", "--"]
CMD ["node", "server.js"]
```

Logging drivers production

```
# Gửi log lên Loki
docker run \
  --log-driver=loki \
  --log-opt loki-url="http://loki:3100/loki/api/v1/push" \
  --log-opt loki-labels="app=myapp,env=prod" \
  myapp

# Gửi log lên fluentd
docker run \
  --log-driver=fluentd \
  --log-opt fluentd-address=localhost:24224 \
  myapp
```

Immutable tags — không dùng latest trong production

```
# Luôn tag với version cụ thể
docker build -t myapp:1.2.3 -t myapp:1.2 .

# Enable Docker Content Trust (signed images)
export DOCKER_CONTENT_TRUST=1
docker pull myapp:1.2.3  # verify signature
```

`latest` không có nghĩa là "mới nhất" — chỉ là default tag khi không chỉ định. Không dùng trong production CI/CD.

04

Kubernetes Tricks: Từ Beginner đến Advanced

> Cập nhật: 2026-05-27 | Phong cách: thực chiến, không lý thuyết thừa

---

1. kubectl Productivity

B Aliases thiết yếu

```
alias k=kubectl
alias kgp='kubectl get pods'
alias kgs='kubectl get svc'
alias kgn='kubectl get nodes'
alias kdp='kubectl describe pod'
alias kl='kubectl logs'
alias kex='kubectl exec -it'
alias kctx='kubectl config use-context'
alias kns='kubectl config set-context --current --namespace'

# Thêm vào ~/.bashrc hoặc ~/.zshrc
```

B Context và namespace switching nhanh

```
# Xem tất cả context
kubectl config get-contexts

# Switch context
kubectl config use-context prod-cluster

# Switch namespace hiện tại (không cần -n mỗi lần)
kubectl config set-context --current --namespace=myapp

# Xem current context/namespace
kubectl config current-context
kubectl config view --minify | grep namespace
```

I krew — package manager cho kubectl plugins

```
# Cài krew
(
  set -x; cd "$(mktemp -d)" &&
  OS="$(uname | tr '[:upper:]' '[:lower:]')" &&
  ARCH="$(uname -m | sed -e 's/x86_64/amd64/' -e 's/arm.*$/arm64/')" &&
  curl -fsSLO "https://github.com/kubernetes-sigs/krew/releases/latest/download/krew-${OS}_$
{ARCH}.tar.gz" &&
  tar zxvf "krew-${OS}_${ARCH}.tar.gz" &&
  KREW=./krew-${OS}_${ARCH} &&
  "$KREW" install krew
)
```

```
# Các plugin hữu ích
kubectl krew install ctx        # switch context nhanh
kubectl krew install ns         # switch namespace nhanh
kubectl krew install neat       # output yaml gọn không có managed fields
kubectl krew install tree       # xem resource hierarchy
kubectl krew install stern      # tail log từ nhiều pod cùng lúc
kubectl krew install node-shell # ssh vào node
```

I dry-run và diff — preview trước khi apply

```
# Xem yaml sẽ được tạo ra (không apply)
kubectl create deployment nginx --image=nginx --dry-run=client -o yaml

# So sánh diff giữa local file và cluster state
kubectl diff -f deployment.yaml
# -  replicas: 1   ← hiện tại trên cluster
# +  replicas: 3   ← trong file local
```

I Output formatting tricks

```
# JSONPath — lấy field cụ thể
kubectl get pods -o jsonpath='{.items[*].metadata.name}'
kubectl get nodes -o jsonpath='{.items[*].status.addresses[?(@.type=="InternalIP")].address}'

# Custom columns
kubectl get pods -o custom-columns='NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName'

# Sort
kubectl get pods --sort-by='.status.startTime'

# Label selector
kubectl get pods -l app=nginx,env=prod
```

A Server-side apply — idempotent deploy

```
# Server-side apply: Kubernetes track field ownership, tránh conflict
kubectl apply --server-side -f deployment.yaml

# Force fieldManager khi conflict
kubectl apply --server-side --force-conflicts -f deployment.yaml
```

! Client-side apply (`kubectl apply`) có thể conflict khi nhiều người/tool cùng quản lý cùng resource.

---

2. Pod Debugging

B Xem logs

```
# Log pod
kubectl logs pod-name

# Log container cụ thể trong multi-container pod
kubectl logs pod-name -c container-name

# Tail live
kubectl logs -f pod-name

# Log của pod đã crash (container trước đó)
```

```
kubectl logs pod-name --previous

# Log từ nhiều pod theo label
kubectl logs -l app=nginx --all-containers=true

# Stern: tail nhiều pod đẹp hơn
stern app=nginx --namespace=prod
```

B Describe — thông tin chi tiết + Events

`kubectl describe` là công cụ đầu tiên cần dùng khi pod có vấn đề. Phần `Events` ở cuối thường giải thích ngay nguyên nhân.

```
kubectl describe pod pod-name
# Events:
#   Warning  FailedScheduling  5s  default-scheduler  0/3 nodes available:
#             3 Insufficient memory.
```

I Port-forward — debug service local

```
# Forward port từ pod
kubectl port-forward pod/pod-name 8080:8080

# Forward từ service (auto chọn pod healthy)
kubectl port-forward svc/myservice 8080:80

# Forward từ deployment
kubectl port-forward deployment/myapp 8080:8080

# Background + log ra file
kubectl port-forward svc/postgres 5432:5432 &
```

I Ephemeral containers — debug distroless/minimal image

Pod không có shell? Attach debug container vào namespace của pod đang chạy.

```
# Kubernetes 1.23+
kubectl debug -it pod-name \
  --image=nicolaka/netshoot \
  --target=app-container

# Hoặc copy pod với thêm debug tools
kubectl debug pod-name -it \
  --image=busybox \
  --copy-to=pod-name-debug \
  --share-processes
```

I Exec vào pod

```
# Shell vào container
kubectl exec -it pod-name -- bash
kubectl exec -it pod-name -- sh  # nếu không có bash

# Chạy lệnh không cần shell tương tác
kubectl exec pod-name -- env
kubectl exec pod-name -- cat /etc/hosts

# Exec vào container cụ thể
kubectl exec -it pod-name -c sidecar-container -- sh
```

A Node shell — ssh vào node không có SSH

```
# Dùng plugin node-shell
kubectl node-shell node-name

# Hoặc tay: privileged pod trên node cụ thể
kubectl run -it --rm debug \
  --image=alpine \
  --restart=Never \
  --overrides='{"spec":{"nodeName":"node-name","hostPID":true,"hostNetwork":true,"containers":
[{"name":"debug","image":"alpine","securityContext":{"privileged":true}}]}}' \
  -- sh
```

! Privileged pod = full access host. Chỉ dùng trong emergency, xóa ngay sau đó.

3. Scheduling

B NodeSelector — chọn node đơn giản

```
spec:
  nodeSelector:
    disktype: ssd
    kubernetes.io/arch: amd64
```

```
# Label node trước
kubectl label node node-1 disktype=ssd
```

I Affinity/Anti-affinity — scheduling linh hoạt

```
spec:
  affinity:
    # Prefer (không bắt buộc) chạy trên node có SSD
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          preference:
            matchExpressions:
              - key: disktype
                operator: In
                values: ["ssd"]

    # Bắt buộc: không chạy 2 pod của app này trên cùng node
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values: ["myapp"]
          topologyKey: kubernetes.io/hostname
```

I Taints và Tolerations — reserve node cho workload cụ thể

```
# Taint node GPU cho chỉ GPU workloads
kubectl taint nodes gpu-node dedicated=gpu:NoSchedule
```

```
# Pod muốn chạy trên node này phải có toleration
```

```yaml
spec:
  tolerations:
    - key: dedicated
      operator: Equal
      value: gpu
      effect: NoSchedule
```

I Topology Spread Constraints — phân phối đều pod

```yaml
spec:
  topologySpreadConstraints:
    - maxSkew: 1                            # lệch tối đa 1 pod giữa các zone
      topologyKey: topology.kubernetes.io/zone
      whenUnsatisfiable: DoNotSchedule
      labelSelector:
        matchLabels:
          app: myapp
```

A Priority Classes — ưu tiên workload quan trọng

```yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
---
spec:
  priorityClassName: high-priority
```

⚠ Pod priority cao có thể preempt (đẩy) pod priority thấp ra khỏi node.

---

4. Resource Management

B Requests vs Limits — quan trọng nhất cần hiểu

- **Requests**: scheduler dùng để chọn node. Container được guarantee ít nhất lượng này.
- **Limits**: container không được vượt quá. Vượt CPU → throttle. Vượt RAM → OOMKilled.

```yaml
resources:
  requests:
    memory: "128Mi"
    cpu: "250m"    # 250 millicores = 0.25 core
  limits:
    memory: "512Mi"
    cpu: "500m"
```

⚠ Không set limits CPU trong nhiều trường hợp production — CPU throttling làm latency tăng đột biến dù node còn tài nguyên.

I QoS Classes — Kubernetes tự assign

QoS Class	Điều kiện	Bị kill khi nào
Guaranteed	requests = limits cho tất cả containers	Cuối cùng
Burstable	requests < limits	Khi node pressure
BestEffort	Không set requests/limits	Đầu tiên

```
kubectl get pod pod-name -o jsonpath='{.status.qosClass}'
# Guaranteed
```

I LimitRange — default resources cho namespace

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: dev
spec:
  limits:
    - default:
        cpu: 500m
        memory: 256Mi
      defaultRequest:
        cpu: 100m
        memory: 128Mi
      type: Container
```

I ResourceQuota — giới hạn tổng namespace

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-quota
  namespace: team-a
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    pods: "50"
```

A VPA — tự động điều chỉnh requests

```
# Cài VPA
kubectl apply -f https://github.com/kubernetes/autoscaler/releases/latest/download/vertical-pod-autoscaler.yaml

# VPA object
kubectl apply -f - <<EOF
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: myapp-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp
```

```
  updatePolicy:
    updateMode: "Off"   # chỉ recommend, không tự apply
EOF

kubectl describe vpa myapp-vpa
# Recommendation:
#   Container: myapp
#     Lower Bound:  cpu: 50m, memory: 64Mi
#     Target:       cpu: 100m, memory: 128Mi
#     Upper Bound:  cpu: 500m, memory: 512Mi
```

---

5. Networking

B Service types

```yaml
# ClusterIP: chỉ accessible trong cluster (mặc định)
spec:
  type: ClusterIP

# NodePort: expose ra ngoài qua port trên mỗi node (30000-32767)
spec:
  type: NodePort
  ports:
    - port: 80
      nodePort: 30080

# LoadBalancer: tạo cloud load balancer (ELB, GLB...)
spec:
  type: LoadBalancer
```

I Headless Service — DNS trực tiếp đến pod IP

Dùng cho StatefulSet, databases, khi cần biết IP từng pod riêng.

```yaml
spec:
  clusterIP: None   # headless
  selector:
    app: postgres
```

```
# DNS resolution: <pod-name>.<service-name>.<namespace>.svc.cluster.local
nslookup postgres-0.postgres.default.svc.cluster.local
```

I NetworkPolicy — firewall trong cluster

Mặc định Kubernetes không có network isolation — mọi pod nói chuyện được với nhau.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-only-frontend
  namespace: production
spec:
  podSelector:
    matchLabels:
      app: api
  policyTypes: [Ingress]
  ingress:
    - from:
        - podSelector:
            matchLabels:
```

```
            app: frontend
    ports:
      - port: 8080
```

! NetworkPolicy chỉ hoạt động nếu CNI plugin hỗ trợ (Calico, Cilium, Weave). Flannel không support.

I Debug DNS trong cluster

```
# Chạy pod debug với network tools
kubectl run dnstest --image=nicolaka/netshoot --rm -it -- bash

# Trong pod:
nslookup kubernetes.default.svc.cluster.local
dig myservice.mynamespace.svc.cluster.local
curl http://myservice.mynamespace.svc.cluster.local

# Xem CoreDNS config
kubectl get configmap coredns -n kube-system -o yaml

# Xem CoreDNS logs
kubectl logs -n kube-system -l k8s-app=kube-dns
```

A ExternalName Service — alias cho external service

```
apiVersion: v1
kind: Service
metadata:
  name: external-db
  namespace: production
spec:
  type: ExternalName
  externalName: rds.amazonaws.com.example.internal
```

App gọi `external-db.production.svc.cluster.local` → CoreDNS CNAME → RDS endpoint.

---

6. Storage

B PV/PVC flow

```
# 1. Admin tạo PersistentVolume (hoặc StorageClass tự tạo)
# 2. Developer tạo PersistentVolumeClaim
# 3. Pod dùng PVC

kubectl get pv                  # Xem tất cả PV
kubectl get pvc                 # Xem PVC trong namespace hiện tại
kubectl describe pvc my-pvc     # Check status: Bound/Pending
```

I StorageClass — dynamic provisioning

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp3
```

```
  iops: "3000"
  throughput: "125"
reclaimPolicy: Delete
allowVolumeExpansion: true
```

```
# PVC dùng StorageClass
spec:
  storageClassName: fast-ssd
  accessModes: [ReadWriteOnce]
  resources:
    requests:
      storage: 10Gi
```

I Volume expansion — mở rộng PVC không downtime

```
# Điều kiện: StorageClass có allowVolumeExpansion: true

# Edit PVC
kubectl edit pvc my-pvc
# Tăng resources.requests.storage từ 10Gi lên 20Gi

# Theo dõi
kubectl get pvc my-pvc -w
# my-pvc   Bound   pvc-abc   20Gi   RWO   fast-ssd   2m
```

A Volume Snapshots

```
# Tạo snapshot
kubectl apply -f - <<EOF
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: mydata-snapshot
spec:
  volumeSnapshotClassName: csi-aws-vsc
  source:
    persistentVolumeClaimName: mydata-pvc
EOF

# Restore từ snapshot
kubectl apply -f - <<EOF
spec:
  dataSource:
    name: mydata-snapshot
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  resources:
    requests:
      storage: 10Gi
EOF
```

---

7. Security

B RBAC — phân quyền cơ bản

```
# Role: quyền trong 1 namespace
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

```
  name: pod-reader
  namespace: dev
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list", "watch"]
---
kind: RoleBinding
metadata:
  name: read-pods-binding
subjects:
  - kind: User
    name: alice
roleRef:
  kind: Role
  name: pod-reader
```

```
# Kiểm tra quyền
kubectl auth can-i list pods --as=alice -n dev
# yes

kubectl auth can-i delete pods --as=alice -n dev
# no
```

I ServiceAccount tokens — không dùng default SA

```
# Tạo SA riêng, chỉ gán quyền cần thiết
apiVersion: v1
kind: ServiceAccount
metadata:
  name: myapp-sa
automountServiceAccountToken: false  # tắt auto-mount
```

```
# Pod chỉ mount token khi thực sự cần
spec:
  serviceAccountName: myapp-sa
  automountServiceAccountToken: true
```

I Pod Security Standards

```
# Label namespace để enforce security standard
kubectl label namespace production \
  pod-security.kubernetes.io/enforce=restricted \
  pod-security.kubernetes.io/warn=restricted

# 3 levels: privileged, baseline, restricted
# restricted: không root, không privilege escalation, read-only rootfs
```

A Secrets management — không lưu secret trong Git

```
# Sealed Secrets: encrypt secret, commit được vào Git
kubeseal --fetch-cert > pub-cert.pem
kubectl create secret generic db-creds \
  --from-literal=password=supersecret \
  --dry-run=client -o yaml | \
  kubeseal --cert pub-cert.pem -o yaml > sealed-secret.yaml

# External Secrets Operator: sync từ AWS SSM / Vault / GCP Secret Manager
kubectl apply -f - <<EOF
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
```

```
metadata:
  name: db-secret
spec:
  secretStoreRef:
    name: aws-ssm
    kind: SecretStore
  target:
    name: db-creds
  data:
    - secretKey: password
      remoteRef:
        key: /prod/db/password
EOF
```

---

8. Configuration

B ConfigMap patterns

```
# Từ file
kubectl create configmap nginx-config --from-file=nginx.conf

# Từ literal
kubectl create configmap app-config \
  --from-literal=DB_HOST=postgres \
  --from-literal=LOG_LEVEL=info

# Mount vào pod
```

```
spec:
  volumes:
    - name: config
      configMap:
        name: nginx-config
  containers:
    - volumeMounts:
        - name: config
          mountPath: /etc/nginx/conf.d
```

I Kustomize — patch không fork

```
# base/kustomization.yaml
resources:
  - deployment.yaml
  - service.yaml

# overlays/prod/kustomization.yaml
resources:                  # 'bases' đã deprecated, dùng 'resources'
  - ../../base
patches:
  - patch.yaml
images:
  - name: myapp
    newTag: v1.2.3

# Apply overlay
kubectl apply -k overlays/prod/

# Preview
kubectl kustomize overlays/prod/ | less
```

I Strategic Merge Patch

```
# Patch deployment không cần apply toàn bộ file
kubectl patch deployment myapp -p '{"spec":{"replicas":5}}'

# Patch với file
kubectl patch deployment myapp --patch-file=patch.yaml
```

```
# patch.yaml — chỉ ghi những gì cần thay đổi
spec:
  template:
    spec:
      containers:
        - name: myapp
          resources:
            limits:
              memory: 1Gi
```

A Helm tricks

```
# Debug template trước khi install
helm template myrelease ./mychart --values=prod-values.yaml | less

# Diff trước khi upgrade (cần plugin helm-diff)
helm diff upgrade myrelease ./mychart --values=prod-values.yaml

# Rollback
helm rollback myrelease 2     # rollback về revision 2
helm history myrelease        # xem lịch sử

# Giữ secrets trong Helm với helm-secrets
helm secrets upgrade myrelease ./mychart \
  -f secrets.yaml.enc \
  -f values.yaml
```

---

9. Cost Optimization

B Phát hiện resource idle

```
# Xem resource usage thực tế vs requests
kubectl top pods --all-namespaces --sort-by=memory

# Node utilization
kubectl top nodes

# Tìm pods không có resource requests
kubectl get pods -A -o json | \
  jq -r '.items[] | select(.spec.containers[].resources.requests == null) |
  .metadata.namespace + "/" + .metadata.name'
```

I Spot/Preemptible nodes — tiết kiệm 60-90%

```
# Node pool riêng cho spot instances (GKE/EKS)
# Taint node spot để chỉ opt-in workload chạy trên đó
kubectl taint nodes spot-node-1 cloud.google.com/gke-spot=true:NoSchedule

# Toleration trong pod (workload chấp nhận bị interrupt)
spec:
  tolerations:
```

```
    - key: cloud.google.com/gke-spot
      operator: Equal
      value: "true"
      effect: NoSchedule
```

I Goldilocks — VPA recommendation dashboard

```
# Cài Goldilocks
helm install goldilocks fairwinds-stable/goldilocks -n goldilocks --create-namespace

# Enable cho namespace
kubectl label namespace production goldilocks.fairwinds.com/enabled=true

# Xem dashboard
kubectl port-forward svc/goldilocks-dashboard 8080:80 -n goldilocks
# Mở http://localhost:8080 → xem recommend cho từng deployment
```

A Bin packing — tắt spread để pack pod dày hơn

```
# Kubernetes 1.22+: dùng bin packing score plugin
# /etc/kubernetes/scheduler-config.yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: bin-packing-scheduler
    pluginConfig:
      - name: NodeResourcesFit
        args:
          scoringStrategy:
            type: MostAllocated   # thay vì LeastAllocated (mặc định)
```

10. Troubleshooting

B CrashLoopBackOff

Pod khởi động, crash, restart liên tục.

```
# Bước 1: xem log của container vừa crash
kubectl logs pod-name --previous

# Bước 2: describe xem exit code
kubectl describe pod pod-name | grep -A5 "Last State"
# Exit Code: 1  ← app crash
# Exit Code: 137 ← OOMKilled
# Exit Code: 139 ← Segfault

# Bước 3: nếu container exit quá nhanh, delay để exec vào
kubectl run debug --image=myapp --command -- sleep infinity
kubectl exec -it debug -- sh
```

B ImagePullBackOff

```
kubectl describe pod pod-name | grep -A10 Events
# Failed to pull image: unauthorized / not found / timeout

# Check image name
kubectl get pod pod-name -o jsonpath='{.spec.containers[*].image}'
```

```
# Check imagePullSecret
kubectl get pod pod-name -o jsonpath='{.spec.imagePullSecrets}'

# Tạo imagePullSecret
kubectl create secret docker-registry regcred \
  --docker-server=registry.example.com \
  --docker-username=myuser \
  --docker-password=mypassword
```

OOMKilled — container hết memory

```
kubectl describe pod pod-name | grep -i oom
# OOMKilled: true
# Exit Code: 137

# Xem memory limit hiện tại
kubectl get pod pod-name -o jsonpath='{.spec.containers[*].resources}'

# Giải pháp ngắn hạn: tăng memory limit
kubectl set resources deployment myapp --limits=memory=1Gi

# Giải pháp đúng: profile memory usage thực tế với VPA
kubectl describe vpa myapp-vpa
```

Stuck Terminating — pod không xóa được

```
# Pod mắc kẹt ở Terminating
kubectl get pods | grep Terminating

# Xem finalizers
kubectl get pod stuck-pod -o jsonpath='{.metadata.finalizers}'

# Xóa finalizers bằng patch
kubectl patch pod stuck-pod -p '{"metadata":{"finalizers":[]}}' --type=merge

# Force delete (cẩn thận: không đợi graceful shutdown)
kubectl delete pod stuck-pod --force --grace-period=0
```

`--force` với stateful workloads (database) có thể gây data corruption. Đảm bảo pod thực sự đã không chạy trên node trước khi force delete.

DNS issues trong cluster

```
# Test DNS từ trong cluster
kubectl run dnstest --image=busybox:1.28 --rm -it -- nslookup kubernetes.default

# Kiểm tra CoreDNS pods
kubectl get pods -n kube-system -l k8s-app=kube-dns

# Xem CoreDNS logs
kubectl logs -n kube-system -l k8s-app=kube-dns --tail=50

# Kiểm tra /etc/resolv.conf trong pod
kubectl exec pod-name -- cat /etc/resolv.conf

# Kiểm tra ndots: nếu service name ngắn bị resolve sai
# /etc/resolv.conf: options ndots:5
# → gọi "db" thực ra thử "db.namespace.svc.cluster.local" trước
```

A Events — timeline sự kiện trong cluster

```
# Xem events sắp xếp theo thời gian
kubectl events --for=pod/pod-name

# Xem events toàn namespace, sort theo time
kubectl get events --sort-by='.lastTimestamp' -n production

# Xem chỉ Warning events
kubectl get events --field-selector type=Warning -n production

# Watch live
kubectl get events -w -n production
```

A Xem tài nguyên thực tế bị throttle

```
# Xem container_cpu_throttled_seconds_total trong Prometheus
# Nếu không có Prometheus, dùng cAdvisor metrics trực tiếp

kubectl top pods --containers
# NAME           CONTAINER   CPU(cores)   MEMORY(bytes)
# myapp-abc123   app         450m         256Mi   ← gần limit 500m → throttle

# Check throttle qua node metrics (cần quyền node access)
kubectl node-shell node-name
cat /sys/fs/cgroup/cpu/kubepods/pod<pod-uid>/cpu.stat | grep throttled
```

05

Nginx Tricks — Từ Cơ Bản Đến Nâng Cao

> Thực chiến, không lý thuyết. Mỗi trick có ví dụ chạy được. Tag: B = beginner, I = intermediate, A = advanced. ! = gotcha.

---

1. Configuration Basics

B Server Block — Cấu hình virtual host

Mỗi domain/app cần một server block riêng trong `/etc/nginx/sites-available/`.

```
server {
    listen 80;
    server_name example.com www.example.com;
    root /var/www/example;
    index index.html index.php;
}
```

Enable bằng symlink:

```
ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled/
nginx -t && systemctl reload nginx
```

B Location Matching Order — Thứ tự ưu tiên khi match URL

Nginx match theo thứ tự ưu tiên sau (không phải theo thứ tự viết trong file):

1. `=` exact match (ưu tiên cao nhất)
2. `^~` prefix match (dừng tìm kiếm nếu match)
3. `~` regex, case-sensitive
4. `~*` regex, case-insensitive
5. Prefix match dài nhất (không có modifier)

```
location = /ping { return 200 "pong"; }          # exact — chỉ /ping
location ^~ /static/ { root /var/www; }           # prefix — /static/* không check regex
location ~* \.(jpg|png|gif)$ { expires 30d; }     # regex — ảnh bất kỳ
location / { try_files $uri $uri/ =404; }         # fallback
```

! `^~` không phải regex — nó chặn regex check nếu prefix match thành công.

B try_files — Fallback file logic

```
# Thử file → thư mục → fallback
location / {
    try_files $uri $uri/ /index.html;
}

# SPA: nếu không có file tĩnh, trả về index.html
```

```
location / {
    try_files $uri $uri/ @fallback;
}
location @fallback {
    proxy_pass http://app_server;
}
```

I alias vs root — Khác nhau quan trọng

`root` nối path gốc + URI. `alias` thay thế hoàn toàn phần URI.

```
# root: /var/www/static + /static/img.png = /var/www/static/static/img.png  ← SAI
location /static/ {
    root /var/www;          # kết quả: /var/www/static/img.png (đúng)
}

# alias: thay /static/ bằng /var/www/files/
location /static/ {
    alias /var/www/files/;  # kết quả: /var/www/files/img.png
}
```

! `alias` phải có dấu `/` cuối nếu location cũng có. `root` thì không cần.

B Reload không downtime

```
nginx -t                    # kiểm tra syntax trước
systemctl reload nginx      # graceful reload, không drop connections
# hoặc
nginx -s reload
```

---

2. Performance Tuning

B worker_processes & worker_connections

```
# /etc/nginx/nginx.conf
worker_processes auto;          # = số CPU cores, tự động detect

events {
    worker_connections 1024;    # max connections per worker
    use epoll;                  # Linux epoll, hiệu quả nhất
    multi_accept on;            # accept nhiều conn cùng lúc
}
```

Tổng max connections = `worker_processes * worker_connections`.

I Keepalive tuning

```
http {
    keepalive_timeout 65;       # giữ conn bao lâu (giây)
    keepalive_requests 1000;    # max requests per keepalive conn

    # Upstream keepalive (quan trọng cho reverse proxy)
    upstream backend {
        server 127.0.0.1:8080;
        keepalive 32;           # pool 32 keepalive connections đến backend
    }

    server {
        location / {
            proxy_pass http://backend;
            proxy_http_version 1.1;
```

```
            proxy_set_header Connection "";   # bắt buộc cho upstream keepalive
        }
    }
}
```

Buffer sizes — Tránh disk I/O khi proxy

```
http {
    # Buffer cho client request body
    client_body_buffer_size 128k;
    client_max_body_size 10m;          # max upload size
    client_header_buffer_size 1k;
    large_client_header_buffers 4 8k;

    # Buffer cho proxy response
    proxy_buffer_size 4k;
    proxy_buffers 8 16k;               # 8 buffers, mỗi cái 16k
    proxy_busy_buffers_size 32k;
}
```

Nếu response lớn hơn tổng buffer, Nginx ghi tạm ra disk — gây chậm.

Gzip compression

```
http {
    gzip on;
    gzip_vary on;
    gzip_min_length 1024;              # không nén file nhỏ hơn 1KB
    gzip_comp_level 6;                 # 1-9, 6 là cân bằng tốt
    gzip_types
        text/plain text/css application/json
        application/javascript text/xml application/xml
        image/svg+xml;
    gzip_proxied any;                  # nén cả response từ upstream
    gzip_disable "msie6";
}
```

Brotli compression (cần module ngx_brotli)

```
# Cài: apt install libnginx-mod-brotli hoặc build từ source
brotli on;
brotli_comp_level 6;
brotli_types text/plain text/css application/json application/javascript;
```

Open file cache — Cache file descriptors

```
http {
    open_file_cache max=10000 inactive=20s;
    open_file_cache_valid 30s;
    open_file_cache_min_uses 2;
    open_file_cache_errors on;
}
```

---

3. Caching

Proxy Cache — Cache response từ upstream

```
http {
    # Khai báo cache zone
    proxy_cache_path /var/cache/nginx
```

```
        levels=1:2
        keys_zone=my_cache:10m      # 10MB metadata
        max_size=10g                 # max disk usage
        inactive=60m                 # xóa nếu không được dùng 60 phút
        use_temp_path=off;           # ghi thẳng vào cache dir, không qua temp

    server {
        location / {
            proxy_cache my_cache;
            proxy_cache_key "$scheme$request_method$host$request_uri";
            proxy_cache_valid 200 1h;
            proxy_cache_valid 404 1m;
            proxy_cache_use_stale error timeout updating;
            add_header X-Cache-Status $upstream_cache_status;  # HIT/MISS/BYPASS
        }
    }
}
```

I Cache bypass — Bỏ qua cache theo điều kiện

```
location / {
    proxy_cache my_cache;

    # Bypass nếu có cookie session hoặc header đặc biệt
    proxy_cache_bypass $cookie_session $http_cache_control;
    proxy_no_cache $cookie_session $http_pragma;
}
```

A Microcaching — Cache 1 giây để chịu traffic spike

```
proxy_cache_path /var/cache/nginx/micro levels=1:2 keys_zone=micro:5m max_size=1g;

server {
    location / {
        proxy_cache micro;
        proxy_cache_valid 200 1s;           # cache 1 giây
        proxy_cache_lock on;                # chỉ 1 request đến backend khi miss
        proxy_cache_lock_timeout 5s;
        proxy_cache_use_stale updating;     # dùng stale trong khi update
    }
}
```

Giảm 99% load trên dynamic page có traffic cao.

A Cache purging — Xóa cache thủ công (module ngx_cache_purge)

```
location ~ /purge(/.*) {
    allow 127.0.0.1;
    deny all;
    proxy_cache_purge my_cache "$scheme$request_method$host$1";
}
```

```
curl -X PURGE http://localhost/purge/path/to/page
```

I FastCGI Cache — Cache PHP response

```
http {
    fastcgi_cache_path /var/cache/nginx/fastcgi
        levels=1:2 keys_zone=php_cache:10m max_size=5g inactive=60m;

    server {
        set $skip_cache 0;
        if ($request_method = POST) { set $skip_cache 1; }
        if ($query_string != "") { set $skip_cache 1; }
```

```
        if ($http_cookie ~* "wordpress_logged_in") { set $skip_cache 1; }

        location ~ \.php$ {
            fastcgi_cache php_cache;
            fastcgi_cache_bypass $skip_cache;
            fastcgi_no_cache $skip_cache;
            fastcgi_cache_valid 200 60m;
            fastcgi_cache_key "$scheme$request_method$host$request_uri";
        }
    }
}
```

---

4. Load Balancing

B Upstream block cơ bản

```
upstream backend {
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
    server 10.0.0.3:8080;
}

server {
    location / {
        proxy_pass http://backend;
    }
}
```

Mặc định là round-robin.

I Các thuật toán load balancing

```
upstream backend {
    least_conn;                          # gửi đến server ít connection nhất

    # hoặc ip_hash — sticky session dựa trên IP client
    # ip_hash;

    # hoặc random (nginx 1.15.1+)
    # random two least_conn;
    # ❗ chỉ bật MỘT phương pháp; không stack least_conn + ip_hash + random

    server 10.0.0.1:8080 weight=3;     # nhận 3x traffic
    server 10.0.0.2:8080 weight=1;
}
```

I Server parameters — Health check thủ công

```
upstream backend {
    server 10.0.0.1:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.2:8080 max_fails=3 fail_timeout=30s;
    server 10.0.0.3:8080 backup;       # chỉ dùng khi tất cả server khác down
}
```

A Active Health Check (Nginx Plus hoặc ngx_upstream_check_module)

```
# Với module nginx_upstream_check_module (open source)
upstream backend {
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
    check interval=3000 rise=2 fall=3 timeout=1000 type=http;
```

```
    check_http_send "HEAD /health HTTP/1.0\r\n\r\n";
    check_http_expect_alive http_2xx;
}
```

I Sticky sessions với cookie

```
# Cần nginx-sticky-module-ng
upstream backend {
    sticky cookie srv_id expires=1h domain=.example.com path=/;
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
}
```

---

5. SSL/TLS

B Certbot + Let's Encrypt

```
certbot --nginx -d example.com -d www.example.com
```

I SSL config tối ưu

```
server {
    listen 443 ssl http2;
    server_name example.com;

    ssl_certificate /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;

    # Chỉ dùng TLS 1.2+ (drop TLS 1.0, 1.1)
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-ECDSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305;
    ssl_prefer_server_ciphers off;      # TLS 1.3 tự chọn cipher

    # Session cache
    ssl_session_cache shared:SSL:10m;
    ssl_session_timeout 1d;
    ssl_session_tickets off;            # forward secrecy
}
```

I OCSP Stapling — Tăng tốc TLS handshake

```
ssl_stapling on;
ssl_stapling_verify on;
ssl_trusted_certificate /etc/letsencrypt/live/example.com/chain.pem;
resolver 1.1.1.1 8.8.8.8 valid=300s;
resolver_timeout 5s;
```

Server tự fetch và đính kèm OCSP response, client không cần check riêng.

A HTTP/3 + QUIC (Nginx 1.25+)

```
server {
    listen 443 ssl;
    listen 443 quic reuseport;          # HTTP/3
    http2 on;

    ssl_certificate ...;
    ssl_certificate_key ...;
```

```
    # Báo cho browser biết có HTTP/3
    add_header Alt-Svc 'h3=":443"; ma=86400';
}
```

⚠ Cần UDP port 443 mở trên firewall.

B Redirect HTTP → HTTPS

```
server {
    listen 80;
    server_name example.com www.example.com;
    return 301 https://$host$request_uri;
}
```

---

6. Security Headers

I Security headers cơ bản

```
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Content-Type-Options "nosniff" always;
add_header X-XSS-Protection "1; mode=block" always;
add_header Referrer-Policy "strict-origin-when-cross-origin" always;
add_header Permissions-Policy "camera=(), microphone=(), geolocation=()" always;
```

I HSTS — Buộc HTTPS

```
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains; preload" always;
```

⚠ Cẩn thận với `preload` — một khi submit vào HSTS preload list rất khó gỡ.

I Content Security Policy

```
add_header Content-Security-Policy "default-src 'self'; script-src 'self' 'nonce-$request_id';
style-src 'self' 'unsafe-inline'; img-src 'self' data: https:; font-src 'self'; connect-src 'self
'; frame-ancestors 'none';" always;
```

I Rate Limiting — Chống brute force và DDoS

```
http {
    # Khai báo zone: 10MB lưu state, 10 req/giây per IP
    limit_req_zone $binary_remote_addr zone=api_limit:10m rate=10r/s;
    limit_req_zone $binary_remote_addr zone=login_limit:10m rate=5r/m;
    limit_conn_zone $binary_remote_addr zone=conn_limit:10m;

    server {
        location /api/ {
            limit_req zone=api_limit burst=20 nodelay;
            limit_conn conn_limit 10;
        }

        location /login {
            limit_req zone=login_limit burst=3;
        }
    }
}
```

A Geo blocking — Chặn theo quốc gia (cần GeoIP2 module)

```
# Cài MaxMind GeoIP2
apt install libnginx-mod-http-geoip2
# Download database: https://dev.maxmind.com/geoip/geolite2-free-geolocation-data
```

```
http {
    geoip2 /usr/share/GeoIP/GeoLite2-Country.mmdb {
        $geoip2_data_country_code country iso_code;
    }

    map $geoip2_data_country_code $allowed_country {
        default no;
        VN yes;
        US yes;
        SG yes;
    }

    server {
        if ($allowed_country = no) {
            return 403;
        }
    }
}
```

B Basic Auth — Bảo vệ endpoint đơn giản

```
htpasswd -c /etc/nginx/.htpasswd username    # tạo file (cần apache2-utils)
htpasswd /etc/nginx/.htpasswd user2          # thêm user
```

```
location /admin {
    auth_basic "Restricted";
    auth_basic_user_file /etc/nginx/.htpasswd;
}
```

I Ẩn version Nginx

```
http {
    server_tokens off;    # không show "nginx/1.24.0" trong header
}
```

---

7. Reverse Proxy

B proxy_pass cơ bản

```
location / {
    proxy_pass http://127.0.0.1:3000;
    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
}
```

I WebSocket proxying

```
location /ws/ {
    proxy_pass http://backend;
    proxy_http_version 1.1;
```

```
    proxy_set_header Upgrade $http_upgrade;       # bắt buộc cho WS
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 3600s;                      # giữ WS conn lâu
    proxy_send_timeout 3600s;
}
```

I Timeout settings

```
location / {
    proxy_pass http://backend;
    proxy_connect_timeout 10s;     # timeout kết nối đến backend
    proxy_send_timeout 60s;        # timeout gửi request đến backend
    proxy_read_timeout 60s;        # timeout chờ response từ backend
}
```

I Tắt buffering cho streaming/SSE

```
location /stream {
    proxy_pass http://backend;
    proxy_buffering off;                    # gửi thẳng đến client
    proxy_cache off;
    proxy_set_header X-Accel-Buffering no;
    add_header X-Accel-Buffering no;
}
```

A X-Accel-Redirect — Backend điều khiển file serving

Backend trả header `X-Accel-Redirect: /internal/file.zip`, Nginx phục vụ file đó từ internal location.

```
location /internal/ {
    internal;                               # chỉ accessible từ Nginx nội bộ
    alias /var/www/protected/;
}
```

Backend code (Python ví dụ):

```
response.headers['X-Accel-Redirect'] = '/internal/secret.pdf'
```

---

8. Logging

B Custom log format

```
http {
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" $request_time';

    access_log /var/log/nginx/access.log main;
    error_log /var/log/nginx/error.log warn;
}
```

I JSON logging — Dễ parse với ELK/Loki

```
log_format json_combined escape=json
    '{'
        '"time":"$time_iso8601",'
        '"remote_addr":"$remote_addr",'
        '"method":"$request_method",'
        '"uri":"$request_uri",'
```

```
        '"status":$status,'
        '"bytes":$body_bytes_sent,'
        '"request_time":$request_time,'
        '"upstream_time":"$upstream_response_time",'
        '"user_agent":"$http_user_agent",'
        '"host":"$host"'
    '}';

access_log /var/log/nginx/access.json json_combined;
```

I Conditional logging — Không log health checks

```
http {
    map $request_uri $loggable {
        default          1;
        /health          0;
        /ping            0;
        ~^/metrics       0;
    }

    access_log /var/log/nginx/access.log combined if=$loggable;
}
```

B Log rotation (logrotate)

```
# /etc/logrotate.d/nginx
/var/log/nginx/*.log {
    daily
    missingok
    rotate 14
    compress
    delaycompress
    notifempty
    sharedscripts
    postrotate
        nginx -s reopen     # hoặc: kill -USR1 $(cat /var/run/nginx.pid)
    endscript
}
```

---

9. Debugging

B Kiểm tra config

```
nginx -t                    # test config, hiển thị lỗi
nginx -T                    # dump toàn bộ config đã parse (bao gồm include)
nginx -T | grep -n keyword  # tìm directive cụ thể
```

B Xem log realtime

```
tail -f /var/log/nginx/error.log
journalctl -u nginx -f      # nếu dùng systemd
```

I stub_status — Metrics cơ bản

```
server {
    listen 127.0.0.1:8080;
    location /nginx_status {
        stub_status;
        allow 127.0.0.1;
        deny all;
```

```
    }
}
```

```
curl http://127.0.0.1:8080/nginx_status
# Active connections: 5
# server accepts handled requests
#  100 100 350
```

I Request ID tracing

```
http {
    map $http_x_request_id $request_id_value {
        default $http_x_request_id;
        ""      $request_id;        # tự sinh nếu không có
    }

    server {
        add_header X-Request-ID $request_id_value always;
        proxy_set_header X-Request-ID $request_id_value;

        log_format trace '$request_id_value $remote_addr "$request" $status';
        access_log /var/log/nginx/trace.log trace;
    }
}
```

B Custom error pages

```
server {
    error_page 404 /404.html;
    error_page 500 502 503 504 /50x.html;

    location = /404.html { root /var/www/errors; internal; }
    location = /50x.html { root /var/www/errors; internal; }
}
```

I Debug log tạm thời

```
error_log /var/log/nginx/debug.log debug;    # rất verbose, chỉ dùng khi debug
```

! `debug` log cực kỳ nặng — không dùng trên production quá 5 phút.

---

10. Advanced

I map directive — Logic điều kiện gọn hơn if

```
http {
    # Map user agent → có phải mobile không
    map $http_user_agent $is_mobile {
        default 0;
        ~*(android|iphone|ipad|mobile) 1;
    }

    # Map origin → CORS allowed
    map $http_origin $cors_header {
        default "";
        "https://app.example.com" "$http_origin";
        "https://admin.example.com" "$http_origin";
    }

    server {
```

```
        add_header Access-Control-Allow-Origin $cors_header always;

        location / {
            if ($is_mobile) {
                return 302 https://m.example.com$request_uri;
            }
        }
    }
}
```

A split_clients — A/B testing

```
http {
    # 20% user thấy version B, 80% thấy version A
    split_clients "${remote_addr}${http_user_agent}" $variant {
        20% "b";
        *   "a";
    }

    server {
        location / {
            proxy_pass http://app_$variant;    # app_a hoặc app_b
        }
    }

    upstream app_a { server 10.0.0.1:3000; }
    upstream app_b { server 10.0.0.2:3000; }
}
```

A njs module — JavaScript trong Nginx

```
apt install nginx-module-njs
```

```
load_module modules/ngx_http_js_module.so;

http {
    js_import main from /etc/nginx/njs/main.js;

    server {
        location /auth {
            js_content main.authenticate;
        }
    }
}
```

```
// /etc/nginx/njs/main.js
function authenticate(r) {
    const token = r.headersIn['Authorization'];
    if (!token || !token.startsWith('Bearer ')) {
        r.return(401, 'Unauthorized\n');
        return;
    }
    r.return(200, 'OK\n');
}
export default { authenticate };
```

A Stream module — TCP/UDP proxy (non-HTTP)

```
# /etc/nginx/nginx.conf — ngoài http block
stream {
    upstream mysql {
        server 10.0.0.1:3306;
        server 10.0.0.2:3306;
    }
```

```
    server {
        listen 3306;
        proxy_pass mysql;
        proxy_timeout 1s;
        proxy_connect_timeout 1s;
    }

    # UDP — DNS load balancing
    upstream dns {
        server 1.1.1.1:53;
        server 8.8.8.8:53;
    }

    server {
        listen 53 udp;
        proxy_pass dns;
    }
}
```

ℹ Snippet reusable với include

```
# /etc/nginx/snippets/ssl-params.conf
ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:...;
ssl_prefer_server_ciphers off;
ssl_session_cache shared:SSL:10m;

# /etc/nginx/snippets/proxy-headers.conf
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;

# Dùng trong server block:
include snippets/ssl-params.conf;
include snippets/proxy-headers.conf;
```

Ⓐ Lua (OpenResty) — Lập trình trong Nginx

```
# Dùng OpenResty thay nginx
apt install openresty
```

```
location /api/check {
    access_by_lua_block {
        local redis = require "resty.redis"
        local red = redis:new()
        red:connect("127.0.0.1", 6379)
        local key = "ratelimit:" .. ngx.var.remote_addr
        local count = red:incr(key)
        if tonumber(count) == 1 then red:expire(key, 60) end
        if tonumber(count) > 100 then
            ngx.exit(429)
        end
    }
    proxy_pass http://backend;
}
```

Quick Reference

```
# Xem cấu hình đầy đủ
nginx -T 2>/dev/null | less

# Reload không downtime
```

```
nginx -s reload

# Kiểm tra open connections
ss -tnp | grep nginx

# Xem process
ps aux | grep nginx

# Test SSL
openssl s_client -connect example.com:443 -servername example.com

# Benchmark nhanh
ab -n 1000 -c 100 https://example.com/
wrk -t4 -c100 -d30s https://example.com/
```

06

Traefik Tricks — Từ Cơ Bản Đến Nâng Cao

> Thực chiến, không lý thuyết. Mỗi trick có ví dụ chạy được. Tag: B = beginner, I = intermediate, A = advanced. ! = gotcha.

---

1. Architecture — Hiểu trước, cấu hình sau

B Các khái niệm cốt lõi

Traefik hoạt động theo mô hình:

```
Internet → EntryPoint → Router → Middleware → Service → Backend
```

Khái niệm	Vai trò
EntryPoint	Cổng lắng nghe (port 80, 443, ...)
Router	Match request theo host/path, gắn middleware
Middleware	Transform request/response (auth, redirect, rate limit)
Service	Định nghĩa backend thực (load balancer, URL)
Provider	Nguồn cấu hình (Docker, File, K8s, Consul)

B File cấu hình cơ bản

Traefik có 2 loại config:

- **Static config** (`traefik.yml` hoặc `traefik.toml`): EntryPoints, providers, logging. **Cần restart khi thay đổi.**
- **Dynamic config**: Routers, middlewares, services. **Hot reload tự động.**

```yaml
# traefik.yml — static config
entryPoints:
  web:
    address: ":80"
  websecure:
    address: ":443"

providers:
  docker:
    exposedByDefault: false    # quan trọng: không expose tất cả container
  file:
    directory: /etc/traefik/dynamic/
    watch: true

api:
  dashboard: true

log:
  level: INFO
```

```
accessLog: {}
```

⚠ Static config và dynamic config là HAI thứ khác nhau. Không trộn lẫn.

---

2. Docker Provider

B Labels cơ bản — Expose một container

```yaml
# docker-compose.yml
services:
  app:
    image: nginx
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.app.rule=Host(`app.example.com`)"
      - "traefik.http.routers.app.entrypoints=websecure"
      - "traefik.http.routers.app.tls.certresolver=letsencrypt"
      - "traefik.http.services.app.loadbalancer.server.port=80"
```

⚠ `traefik.http.services.app.loadbalancer.server.port` phải là port **trong container**, không phải port expose ra host.

B Network config — Container phải cùng network với Traefik

```yaml
# docker-compose.yml
services:
  traefik:
    image: traefik:v3.0
    networks:
      - traefik_net
    ports:
      - "80:80"
      - "443:443"

  app:
    image: myapp
    networks:
      - traefik_net           # phải kết nối vào đây
    labels:
      - "traefik.enable=true"
      - "traefik.docker.network=traefik_net"   # chỉ rõ network nếu app có nhiều network

networks:
  traefik_net:
    external: true
```

```
docker network create traefik_net
```

⚠ Nếu container có nhiều network, **bắt buộc** dùng `traefik.docker.network` để Traefik biết dùng network nào.

I Auto-discovery với constraints

```yaml
# traefik.yml
providers:
  docker:
    exposedByDefault: false
    constraints: "Label(`traefik.env`,`production`)"   # chỉ expose container có label này
```

```
# Chỉ container này mới được Traefik nhận
labels:
  - "traefik.enable=true"
  - "traefik.env=production"
```

I defaultRule — Tự động route theo container name

```
# traefik.yml
providers:
  docker:
    defaultRule: "Host(`{{ normalize .Name }}.example.com`)"
    exposedByDefault: true
```

Container tên `api` sẽ tự động route đến `api.example.com` không cần label.

! `normalize` chuyển underscores và ký tự đặc biệt thành dash. Container `my_app` → `my-app.example.com`.

I Traefik với Docker Swarm

```
# traefik.yml
providers:
  swarm:
    endpoint: "unix:///var/run/docker.sock"
    exposedByDefault: false
    watch: true
```

```
# Deploy Traefik như global service trong Swarm
docker service create \
  --name traefik \
  --constraint "node.role==manager" \
  --publish 80:80 --publish 443:443 \
  --mount type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
  traefik:v3.0
```

---

3. File Provider

B Dynamic config từ file YAML

```
# /etc/traefik/dynamic/services.yml
http:
  routers:
    my-router:
      rule: "Host(`example.com`)"
      entryPoints:
        - websecure
      service: my-service
      tls:
        certResolver: letsencrypt

  services:
    my-service:
      loadBalancer:
        servers:
          - url: "http://10.0.0.1:3000"
          - url: "http://10.0.0.2:3000"
```

I Directory watching — Chia nhỏ config

```
# traefik.yml
providers:
```

```
  file:
    directory: /etc/traefik/dynamic/
    watch: true       # hot reload khi file thay đổi
```

Cấu trúc:

```
/etc/traefik/dynamic/
├── routers.yml
├── middlewares.yml
├── services.yml
└── tls.yml
```

⚠ Tất cả file trong directory đều được load. Trùng tên router/service → conflict.

ℹ TOML vs YAML — Chọn nhất quán

```
# TOML — cú pháp chặt hơn, ít lỗi indent
[http.routers.my-router]
  rule = "Host(`example.com`)"
  entryPoints = ["websecure"]
  service = "my-service"

[http.services.my-service.loadBalancer]
  [[http.services.my-service.loadBalancer.servers]]
    url = "http://10.0.0.1:3000"
```

Nên chọn 1 format và dùng xuyên suốt. Không mix YAML + TOML trong cùng directory.

---

4. TLS & Let's Encrypt

B ACME HTTP Challenge

```
# traefik.yml
certificatesResolvers:
  letsencrypt:
    acme:
      email: admin@example.com
      storage: /etc/traefik/acme.json     # lưu certificates
      httpChallenge:
        entryPoint: web                   # challenge qua port 80
```

```
# Tạo file và set quyền đúng
touch /etc/traefik/acme.json
chmod 600 /etc/traefik/acme.json         # bắt buộc 600, Traefik từ chối nếu khác
```

⚠ `acme.json` phải có permission `600`. Nếu không Traefik log lỗi và không lưu cert.

ℹ DNS Challenge — Wildcard certificates

```
# traefik.yml
certificatesResolvers:
  letsencrypt:
    acme:
      email: admin@example.com
      storage: /etc/traefik/acme.json
      dnsChallenge:
        provider: cloudflare              # hoặc: route53, digitalocean, godaddy...
        resolvers:
          - "1.1.1.1:53"
          - "8.8.8.8:53"
```

```
# docker-compose.yml — env vars cho Cloudflare
environment:
  - CF_API_EMAIL=admin@example.com
  - CF_API_KEY=your-api-key
  # hoặc dùng token:
  - CF_DNS_API_TOKEN=your-token
```

```
# Label cho wildcard cert
labels:
  - "traefik.http.routers.app.tls.certresolver=letsencrypt"
  - "traefik.http.routers.app.tls.domains[0].main=example.com"
  - "traefik.http.routers.app.tls.domains[0].sans=*.example.com"
```

Default certificate — Cert fallback khi không match

```
# dynamic/tls.yml
tls:
  stores:
    default:
      defaultCertificate:
        certFile: /certs/default.crt
        keyFile: /certs/default.key
```

Multiple cert resolvers

```
# traefik.yml — staging để test, production để deploy
certificatesResolvers:
  staging:
    acme:
      caServer: https://acme-staging-v02.api.letsencrypt.org/directory
      email: admin@example.com
      storage: /etc/traefik/acme-staging.json
      httpChallenge:
        entryPoint: web

  production:
    acme:
      email: admin@example.com
      storage: /etc/traefik/acme-prod.json
      httpChallenge:
        entryPoint: web
```

Luôn test với `staging` trước — Let's Encrypt production giới hạn 50 cert/registered domain/tuần (và 5 cert trùng/tuần).

---

5. Middleware Chains

Redirect HTTP → HTTPS

```
# dynamic/middlewares.yml
http:
  middlewares:
    redirect-to-https:
      redirectScheme:
        scheme: https
        permanent: true
```

```
# Router dùng middleware
http:
  routers:
```

```
    app-http:
      rule: "Host(`example.com`)"
      entryPoints:
        - web
      middlewares:
        - redirect-to-https
      service: app-service
```

B Headers middleware — Security headers

```
http:
  middlewares:
    secure-headers:
      headers:
        frameDeny: true
        contentTypeNosniff: true
        browserXssFilter: true
        forceSTSHeader: true
        stsSeconds: 31536000
        stsIncludeSubdomains: true
        stsPreload: true
        customResponseHeaders:
          X-Robots-Tag: "noindex,nofollow,nosnippet,noarchive"
          server: ""                    # ẩn server header
```

I BasicAuth middleware

```
# Tạo password hash (cần htpasswd)
htpasswd -nb admin password | sed -e 's/\$/\$\$/g'
# Output: admin:$$apr1$$...  (escape $ cho YAML)
```

```
http:
  middlewares:
    auth:
      basicAuth:
        users:
          - "admin:$$apr1$$xyz$$hashedpassword"
        removeHeader: true    # xóa Authorization header trước khi pass đến backend
```

I RateLimit middleware

```
http:
  middlewares:
    rate-limit:
      rateLimit:
        average: 100          # 100 req/giây
        burst: 50             # cho phép burst 50 req
        period: 1s
        sourceCriterion:
          ipStrategy:
            depth: 1          # lấy IP thực từ X-Forwarded-For
```

I stripPrefix — Bỏ prefix khi proxy

```
http:
  middlewares:
    strip-api:
      stripPrefix:
        prefixes:
          - "/api"

  routers:
    api:
      rule: "Host(`example.com`) && PathPrefix(`/api`)"
```

```
      middlewares:
        - strip-api
      service: api-service

# Request: /api/users → backend nhận: /users
```

I Chain — Gộp nhiều middleware

```
http:
  middlewares:
    secured:
      chain:
        middlewares:
          - redirect-to-https
          - secure-headers
          - rate-limit
```

```
# Dùng chain trong router
labels:
  - "traefik.http.routers.app.middlewares=secured@file"
```

A ForwardAuth — Delegate auth đến external service

```
http:
  middlewares:
    oauth:
      forwardAuth:
        address: "http://auth-service:4181"
        trustForwardHeader: true
        authResponseHeaders:
          - "X-Auth-User"
          - "X-Auth-Email"
```

Traefik gửi request đến `auth-service`, nếu trả về 200 thì cho qua, 401 thì block.

---

6. Load Balancing

B Round robin cơ bản

```
http:
  services:
    my-service:
      loadBalancer:
        servers:
          - url: "http://10.0.0.1:3000"
          - url: "http://10.0.0.2:3000"
          - url: "http://10.0.0.3:3000"
```

I Weighted Round Robin

```
http:
  services:
    weighted-service:
      weighted:
        services:
          - name: v1
            weight: 80
          - name: v2
            weight: 20
```

```
    v1:
      loadBalancer:
        servers:
          - url: "http://10.0.0.1:3000"

    v2:
      loadBalancer:
        servers:
          - url: "http://10.0.0.2:3000"
```

I Sticky sessions

```
http:
  services:
    my-service:
      loadBalancer:
        sticky:
          cookie:
            name: traefik_sticky
            secure: true
            httpOnly: true
            sameSite: strict
        servers:
          - url: "http://10.0.0.1:3000"
          - url: "http://10.0.0.2:3000"
```

I Health checks

```
http:
  services:
    my-service:
      loadBalancer:
        healthCheck:
          path: /health
          interval: 10s
          timeout: 3s
          scheme: http
          followRedirects: false
          headers:
            X-Health-Check: "traefik"
        servers:
          - url: "http://10.0.0.1:3000"
          - url: "http://10.0.0.2:3000"
```

A Mirroring — Duplicate traffic để test

```
http:
  services:
    main-with-mirror:
      mirroring:
        service: main          # traffic chính
        maxBodySize: 1048576   # 1MB max để mirror
        mirrors:
          - name: shadow
            percent: 10        # 10% traffic copy đến shadow
```

Dùng để test version mới với traffic thật mà không ảnh hưởng user.

---

7. Dashboard & API

B Enable Dashboard an toàn

```yaml
# traefik.yml
api:
  dashboard: true
  insecure: false     # KHÔNG bao giờ để true trên production
```

```yaml
# dynamic/dashboard.yml
http:
  routers:
    dashboard:
      rule: "Host(`traefik.example.com`) && (PathPrefix(`/api`) || PathPrefix(`/dashboard`))"
      entryPoints:
        - websecure
      middlewares:
        - dashboard-auth
      service: api@internal    # service đặc biệt, không cần khai báo
      tls:
        certResolver: letsencrypt

  middlewares:
    dashboard-auth:
      basicAuth:
        users:
          - "admin:$$apr1$$hashedpassword"
```

! `service: api@internal` là keyword đặc biệt — không thay thế bằng tên khác.

I Read-only API access

```yaml
# traefik.yml
api:
  dashboard: true
  debug: false      # tắt debug endpoint trên production
```

```bash
# Query API
curl http://traefik.example.com/api/http/routers      # list routers
curl http://traefik.example.com/api/http/services     # list services
curl http://traefik.example.com/api/http/middlewares  # list middlewares
curl http://traefik.example.com/api/overview          # tổng quan
```

8. Observability

B Access logs

```yaml
# traefik.yml
accessLog:
  filePath: /var/log/traefik/access.log
  format: json            # hoặc common (CLF format)
  bufferingSize: 100      # buffer 100 lines trước khi ghi disk
  filters:
    statusCodes:
      - "400-599"         # chỉ log lỗi
    retryAttempts: true
    minDuration: "100ms"  # chỉ log request chậm hơn 100ms
```

B Prometheus metrics

```yaml
# traefik.yml
metrics:
  prometheus:
    addEntryPointsLabels: true
    addRoutersLabels: true
    addServicesLabels: true
    buckets:
      - 0.1
      - 0.3
      - 1.2
      - 5.0
    entryPoint: metrics      # expose trên entrypoint riêng

entryPoints:
  metrics:
    address: ":8082"
```

```yaml
# Prometheus scrape config
scrape_configs:
  - job_name: traefik
    static_configs:
      - targets: ["traefik:8082"]
```

```bash
# Xem metrics thô
curl http://traefik:8082/metrics | grep traefik_
```

I Tracing với Jaeger/Zipkin

! Traefik v3 đã bỏ tracing native cho Jaeger/Zipkin — chỉ còn **OTLP** (OpenTelemetry). Muốn đẩy trace sang Jaeger thì gửi qua OTLP collector.

```yaml
# traefik.yml — OTLP (chuẩn Traefik v3)
tracing:
  otlp:
    grpc:
      endpoint: http://otel-collector:4317
      insecure: true
```

I Structured logging

```yaml
# traefik.yml
log:
  level: INFO              # DEBUG, INFO, WARN, ERROR
  format: json             # hoặc common
  filePath: /var/log/traefik/traefik.log
  noColor: true            # tắt ANSI color khi log vào file
```

---

9. Advanced Patterns

I TCP routing — Non-HTTP protocols

```yaml
# dynamic/tcp.yml
tcp:
  routers:
    postgres:
      rule: "HostSNI(`db.example.com`)"    # cần SNI (TLS)
      entryPoints:
        - postgres                          # port 5432
      service: postgres-service
```

```
      tls:
        passthrough: true                    # TLS passthrough, không terminate

    # Không có TLS — route theo entrypoint
    raw-tcp:
      rule: "HostSNI(`*`)"
      entryPoints:
        - raw                                # port 9000
      service: raw-service

  services:
    postgres-service:
      loadBalancer:
        servers:
          - address: "10.0.0.1:5432"

    raw-service:
      loadBalancer:
        servers:
          - address: "10.0.0.1:9000"
```

```
# traefik.yml — thêm entrypoints cho TCP
entryPoints:
  postgres:
    address: ":5432"
  raw:
    address: ":9000"
```

UDP routing

```
# dynamic/udp.yml
udp:
  routers:
    dns:
      entryPoints:
        - dns
      service: dns-service

  services:
    dns-service:
      loadBalancer:
        servers:
          - address: "10.0.0.1:53"
```

InFlightReq — Giới hạn concurrent requests

```
http:
  middlewares:
    limit-concurrent:
      inFlightReq:
        amount: 10                   # max 10 request đồng thời
        sourceCriterion:
          requestHeaderName: "X-User-ID"   # giới hạn per user
          # hoặc:
          ipStrategy:
            depth: 1
```

CircuitBreaker

```
http:
  middlewares:
    circuit-breaker:
      circuitBreaker:
        expression: "NetworkErrorRatio() > 0.30 || ResponseCodeRatio(500, 600, 0, 600) > 0.25"
        # Mở circuit nếu: >30% network error HOẶC >25% 5xx response
        checkPeriod: 10s
```

```
        fallbackDuration: 30s        # giữ open 30 giây
        recoveryDuration: 10s        # half-open để test
```

A Retry

```
http:
  middlewares:
    retry:
      retry:
        attempts: 3
        initialInterval: 100ms       # backoff bắt đầu
```

! Retry chỉ an toàn với **idempotent requests** (GET, HEAD). Dùng với POST có thể gây duplicate.

A Canary deployment với weighted service

```
http:
  services:
    canary:
      weighted:
        services:
          - name: stable
            weight: 95
          - name: canary
            weight: 5

    stable:
      loadBalancer:
        servers:
          - url: "http://app-stable:3000"

    canary:
      loadBalancer:
        servers:
          - url: "http://app-canary:3000"
```

Tăng dần `weight` của canary khi đã verify OK.

---

10. Production

B Graceful shutdown

```
# traefik.yml
global:
  sendAnonymousUsage: false

entryPoints:
  web:
    address: ":80"
    transport:
      lifeCycle:
        graceTimeOut: 30s      # đợi 30s để drain connections khi shutdown
        requestAcceptGraceTimeout: 5s
```

```
# Graceful restart
docker kill --signal=SIGUSR1 traefik    # graceful reload config
docker kill --signal=SIGTERM traefik    # graceful shutdown
```

HA với Redis — Chia sẻ ACME certificates

```
# traefik.yml — cho multiple Traefik instances
certificatesResolvers:
  letsencrypt:
    acme:
      storage: "redis://redis:6379"     # thay vì file
      email: admin@example.com
      tlsChallenge: {}
```

Khi dùng Redis storage, tất cả Traefik instances phải trỏ vào cùng Redis để tránh race condition khi renew cert.

HA với Consul

```
# traefik.yml
providers:
  consul:
    endpoints:
      - "consul:8500"
    rootKey: "traefik"

certificatesResolvers:
  letsencrypt:
    acme:
      storage: "consul://consul:8500/traefik/acme"
```

Pilot/Hub connection (Traefik Hub)

```
# traefik.yml
hub:
  token: "your-hub-token"        # Traefik Hub để quản lý tập trung
```

Kubernetes IngressRoute

```
# k8s CRD — thay vì Ingress standard
apiVersion: traefik.io/v1alpha1
kind: IngressRoute
metadata:
  name: my-app
spec:
  entryPoints:
    - websecure
  routes:
    - match: Host(`app.example.com`)
      kind: Rule
      services:
        - name: my-service
          port: 80
      middlewares:
        - name: secure-headers
  tls:
    certResolver: letsencrypt
```

```
# Middleware CRD
apiVersion: traefik.io/v1alpha1
kind: Middleware
metadata:
  name: secure-headers
spec:
  headers:
    frameDeny: true
    contentTypeNosniff: true
```

Rate limit HA với Redis (Traefik Enterprise)

```yaml
http:
  middlewares:
    global-rate-limit:
      rateLimit:
        average: 1000
        burst: 500
        # Community edition: per-instance limits
        # Enterprise: distributed rate limiting via Redis
```

Entrypoint forwarding — ProxyProtocol

```yaml
# Khi đứng sau AWS ALB / GCP LB truyền ProxyProtocol
entryPoints:
  web:
    address: ":80"
    proxyProtocol:
      trustedIPs:
        - "10.0.0.0/8"
        - "172.16.0.0/12"
    forwardedHeaders:
      trustedIPs:
        - "10.0.0.0/8"
```

Quick Reference

```bash
# Xem logs realtime
docker logs -f traefik

# Validate config (Traefik v3)
traefik healthcheck --ping

# List routers qua API
curl -s http://localhost:8080/api/http/routers | jq '.[].name'

# List services
curl -s http://localhost:8080/api/http/services | jq '.[].name'

# Kiểm tra cert
curl -s http://localhost:8080/api/tls/certificates | jq '.[].subject'

# Xem overview
curl -s http://localhost:8080/api/overview | jq .

# Ping health check
curl http://localhost:8080/ping
```

Label cheat sheet cho Docker

```yaml
labels:
  # Bật Traefik
  - "traefik.enable=true"

  # Router
  - "traefik.http.routers.myapp.rule=Host(`app.example.com`)"
  - "traefik.http.routers.myapp.entrypoints=websecure"
  - "traefik.http.routers.myapp.tls.certresolver=letsencrypt"

  # Service (khi container expose nhiều port)
  - "traefik.http.services.myapp.loadbalancer.server.port=3000"

  # Middleware gắn vào router
```

```
  - "traefik.http.routers.myapp.middlewares=auth@file,rate-limit@file"

  # Tạo middleware inline qua label
  - "traefik.http.middlewares.myapp-auth.basicauth.users=admin:$$hash"
  - "traefik.http.routers.myapp.middlewares=myapp-auth"
```

⚠ Middleware trong label dùng cú pháp `name@provider`: `auth@file`, `auth@docker`, `auth@kubernetes`.

07

Redis Tricks — Beginner to Advanced

> Giải thích bằng tiếng Việt, lệnh bằng tiếng Anh. Tags: B beginner · I intermediate · A advanced · ! gotcha

---

1. Data Structures Beyond Strings

B Hash — lưu object thay vì JSON string

Thay vì lưu cả object JSON vào một string key, dùng Hash để truy cập/cập nhật từng field riêng lẻ mà không cần deserialize toàn bộ.

```
HSET user:1001 name "Hieu" age 28 city "HCM"
HGET user:1001 name          # → "Hieu"
HMGET user:1001 name city    # → ["Hieu", "HCM"]
HINCRBY user:1001 age 1      # tăng age lên 1, atomic
HGETALL user:1001
```

Khi nào dùng: user profile, session data, config object. Tránh dùng khi field count > vài nghìn hoặc value rất lớn.

B List — queue và stack

List là linked list hai đầu. Dùng LPUSH+RPOP cho queue FIFO, LPUSH+LPOP cho stack.

```
LPUSH jobs "send_email" "resize_image"
RPOP jobs           # → "send_email" (FIFO)
LLEN jobs           # → 1
LRANGE jobs 0 -1    # xem toàn bộ

# Blocking pop — worker chờ job, không cần polling
BLPOP jobs 30       # chờ tối đa 30s
```

! LRANGE O(N) — không dùng trên list dài để lấy toàn bộ.

B Set — unique collection, quan hệ tập hợp

Không có thứ tự, không trùng lặp. Dùng cho tag, follower list, online users.

```
SADD online_users 101 202 303
SISMEMBER online_users 101   # → 1 (có)
SCARD online_users           # → 3 (đếm)

# Quan hệ tập hợp — ví dụ common friends
SADD user:1:friends 10 20 30
SADD user:2:friends 20 30 40
SINTER user:1:friends user:2:friends   # → [20, 30]
```

```
SUNION user:1:friends user:2:friends   # → [10, 20, 30, 40]
SDIFF  user:1:friends user:2:friends   # → [10]
```

Sorted Set (ZSet) — leaderboard, priority queue

Mỗi member gắn một score float. Redis tự sort theo score. O(log N) cho add/remove.

```
ZADD leaderboard 1500 "alice" 2300 "bob" 980 "carol"
ZRANGE leaderboard 0 -1 WITHSCORES REV    # top-down
ZRANK leaderboard "bob"                   # rank (0-indexed)
ZINCRBY leaderboard 100 "alice"           # tăng score
ZRANGEBYSCORE leaderboard 1000 2000       # lọc theo range score
ZREMRANGEBYRANK leaderboard 0 -101        # giữ top 100
```

Khi nào dùng: leaderboard, rate limiter sliding window, delayed job queue (score = unix timestamp).

Streams — event log bền vững

Stream giống Kafka mini trong Redis. Dữ liệu persist, consumer group hỗ trợ at-least-once delivery.

```
# Producer ghi event
XADD events * action "login" user_id "1001" ip "1.2.3.4"
# → "1748300000000-0" (auto ID = timestamp-sequence)

# Consumer đọc từ đầu
XREAD COUNT 10 STREAMS events 0

# Consumer Group — multiple worker cùng xử lý
XGROUP CREATE events workers $ MKSTREAM
XREADGROUP GROUP workers worker-1 COUNT 5 BLOCK 2000 STREAMS events >
XACK events workers 1748300000000-0    # xác nhận đã xử lý
```

---

2. Bitmap & HyperLogLog

Bitmap — DAU tracking, feature flags

Mỗi bit = 1 user. 100 triệu user chỉ tốn ~12MB.

```
# User 1001 login ngày 2026-05-27
SETBIT dau:2026-05-27 1001 1
GETBIT dau:2026-05-27 1001    # → 1
BITCOUNT dau:2026-05-27       # tổng user active hôm nay

# Feature flag: bật tính năng cho user 1001
SETBIT feature:dark_mode 1001 1
```

BITOP — retention analysis

Tìm user active CẢ 3 ngày liên tiếp (day-1 AND day-2 AND day-3):

```
BITOP AND retained:3day dau:2026-05-25 dau:2026-05-26 dau:2026-05-27
BITCOUNT retained:3day    # số user active đủ 3 ngày
```

BITOP là O(N) theo size bitmap — chạy offline, không dùng trong hot path.

I HyperLogLog — đếm unique với sai số ~0.81%

Đếm unique visitors mà không tốn bộ nhớ (tối đa 12KB bất kể cardinality).

```
PFADD uv:2026-05-27 "user:101" "user:202" "user:101"
PFCOUNT uv:2026-05-27    # → 2 (deduplicated)

# Merge nhiều ngày
PFMERGE uv:week uv:2026-05-21 uv:2026-05-22 uv:2026-05-27
PFCOUNT uv:week
```

Khi nào dùng: unique page views, funnel unique users. Không dùng khi cần chính xác 100%.

---

3. Pub/Sub & Streams

B Pub/Sub — fire-and-forget messaging

Không lưu message, subscriber phải online mới nhận được.

```
# Terminal 1 — subscriber
SUBSCRIBE notifications

# Terminal 2 — publisher
PUBLISH notifications '{"type":"order","id":42}'

# Pattern subscribe
PSUBSCRIBE order:*    # nhận tất cả channel bắt đầu bằng "order:"
```

! Pub/Sub không persist. Nếu subscriber offline → mất message. Dùng Streams thay thế cho use case cần reliability.

A Consumer Groups với dead letter pattern

```
# Đọc pending messages (chưa ACK quá 60s) để xử lý lại
XPENDING events workers - + 10

# Claim lại message từ worker đã chết
XCLAIM events workers worker-2 60000 1748300000000-0

# Trim stream giữ 10000 entries gần nhất
XADD events MAXLEN ~ 10000 * action "click"
XTRIM events MAXLEN ~ 10000
```

---

4. Lua Scripting

I EVAL — atomic operations

Lua script chạy atomic trong Redis, không bị interrupt.

```
# Tăng counter chỉ khi < limit (atomic check-and-increment)
EVAL "
  local current = redis.call('GET', KEYS[1])
  if current == false then current = 0 end
  if tonumber(current) < tonumber(ARGV[1]) then
    return redis.call('INCR', KEYS[1])
  else
    return -1
```

```
  end
" 1 counter:api 100
```

A Rate limiter script — sliding window

```lua
-- rate_limiter.lua
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local now = tonumber(ARGV[3])

redis.call('ZREMRANGEBYSCORE', key, 0, now - window)
local count = redis.call('ZCARD', key)
if count < limit then
  redis.call('ZADD', key, now, now)
  redis.call('EXPIRE', key, window)
  return 1   -- allowed
else
  return 0   -- rejected
end
```

```
EVAL "$(cat rate_limiter.lua)" 1 rate:user:1001 100 60 $(date +%s%3N)
```

A Distributed lock script

```
# Acquire lock — SET NX PX là atomic
SET lock:resource "unique-token-abc" NX PX 30000

# Release lock — chỉ release nếu token khớp (dùng Lua để atomic)
EVAL "
  if redis.call('GET', KEYS[1]) == ARGV[1] then
    return redis.call('DEL', KEYS[1])
  else
    return 0
  end
" 1 lock:resource unique-token-abc
```

! Không dùng GET rồi DEL riêng lẻ — race condition. Phải dùng Lua.

I Debug Lua script

```
redis-cli --ldb EVAL "$(cat script.lua)" 1 mykey myarg
# Trong debug session:
# s → step, c → continue, p var → print variable
```

5. Memory Optimization

I Encoding thresholds — ziplist/listpack

Redis tự động dùng compact encoding khi collection nhỏ. Vượt threshold → chuyển sang encoding tốn RAM hơn.

```
OBJECT ENCODING myhash    # → "ziplist" hoặc "listpack" hoặc "hashtable"
OBJECT ENCODING myzset    # → "listpack" hoặc "skiplist"

# Xem config threshold
```

```
CONFIG GET hash-max-listpack-entries   # mặc định 128
CONFIG GET hash-max-listpack-value     # mặc định 64 bytes
CONFIG GET zset-max-listpack-entries   # mặc định 128
```

Tối ưu: Giữ collection dưới threshold nếu RAM là concern. Ví dụ chia user hash thành nhiều bucket nhỏ.

I MEMORY USAGE — tìm key tốn RAM

```
MEMORY USAGE user:1001           # bytes của key này
MEMORY USAGE user:1001 SAMPLES 5 # sample nested structures

# Tìm top big keys
redis-cli --bigkeys              # scan toàn bộ, in top 5 per type
redis-cli --memkeys              # tương tự nhưng sort theo memory
```

I Eviction policies

```
CONFIG SET maxmemory 2gb
CONFIG SET maxmemory-policy allkeys-lru
# Các policies:
# noeviction      → error khi full (default, dùng cho persistent data)
# allkeys-lru     → evict least recently used (cache thuần)
# volatile-lru    → evict LRU chỉ key có TTL (cache + persistent mixed)
# allkeys-lfu     → evict least frequently used (Redis 4+)
# volatile-ttl    → evict key gần expire nhất
```

! Với cache-only Redis, dùng `allkeys-lru` hoặc `allkeys-lfu`. Không bao giờ dùng `noeviction` cho cache.

---

6. Persistence

B RDB vs AOF so sánh nhanh

	RDB	AOF
Recovery	nhanh	chậm hơn (replay log)
Data loss	tối đa vài phút	tối đa 1 giây (fsync=everysec)
File size	nhỏ (compressed)	lớn hơn, cần rewrite
Fork overhead	có khi snapshot	ít hơn

I Cấu hình persistence

```
# RDB: save sau 60s nếu có ít nhất 1000 key thay đổi
CONFIG SET save "3600 1 300 100 60 1000"

# AOF với fsync mỗi giây (balance giữa perf và durability)
CONFIG SET appendonly yes
CONFIG SET appendfsync everysec

# Dùng cả hai: RDB cho fast restart, AOF cho durability
```

A RDB-on-replica pattern

Tắt RDB trên master để tránh fork overhead, bật trên replica:

```
# Master config
save ""                    # tắt RDB
appendonly yes             # bật AOF

# Replica config
save "900 1 300 10"        # bật RDB trên replica
replicaof master-ip 6379
```

A Fork overhead — BGSAVE impact

```
INFO persistence
# → rdb_last_bgsave_time_sec: thời gian BGSAVE gần nhất
# → aof_rewrite_in_progress: 1 nếu đang rewrite

# Manual trigger
BGSAVE           # async RDB snapshot
BGREWRITEAOF     # compact AOF file
```

! BGSAVE fork() toàn bộ memory — trên server 32GB Redis, fork có thể mất 1-2s, gây latency spike. Monitor `latest_fork_usec` trong `INFO stats`.

---

7. Replication & Cluster

B Sentinel setup — HA cho single shard

```
# sentinel.conf
sentinel monitor mymaster 127.0.0.1 6379 2   # quorum = 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 60000

redis-sentinel /etc/redis/sentinel.conf
```

I READONLY replicas — đọc từ replica

```
# Kết nối vào replica
redis-cli -h replica-host -p 6379
READONLY        # bật read-only mode (mặc định trong cluster)

# Application: route read queries đến replica
# redis-py: với decode_responses=True, dùng Redis(read_from_replicas=True)
```

A Cluster sharding & resharding

```
# Tạo cluster 3 master + 3 replica
redis-cli --cluster create \
  node1:6379 node2:6379 node3:6379 \
  node4:6379 node5:6379 node6:6379 \
  --cluster-replicas 1

# Xem slot distribution
redis-cli -c CLUSTER SHARDS
```

```
# Reshard 1000 slots từ node A sang node B
redis-cli --cluster reshard node1:6379 \
  --cluster-from <source-node-id> \
  --cluster-to <dest-node-id> \
  --cluster-slots 1000 \
  --cluster-yes
```

❗ Key phải có `{hashtag}` nếu cần multi-key ops trong cluster: `user:{1001}:profile`, `user:{1001}:settings` → cùng slot.

---

8. Key Management

B SCAN thay vì KEYS

```
# KEYS là O(N) blocking — KHÔNG dùng trên production
KEYS user:*          # ❗ blocks Redis trong khi scan

# SCAN an toàn hơn — iterative, non-blocking
SCAN 0 MATCH "user:*" COUNT 100
# → [next_cursor, [keys...]]
# Lặp cho đến khi cursor = 0

# Scan theo type
SCAN 0 MATCH "session:*" COUNT 100 TYPE string
```

I Key expiration patterns

```
# Set TTL khi tạo
SET session:abc "data" EX 3600         # expire sau 1h
SET session:abc "data" EXAT 1748400000 # expire tại unix timestamp

# Set TTL sau
EXPIRE key 3600
EXPIREAT key 1748400000
PERSIST key    # xóa TTL

# Xem TTL còn lại
TTL key        # giây (-1 = no expire, -2 = không tồn tại)
PTTL key       # milliseconds
```

I Keyspace notifications — event-driven expiry

```
# Bật notification cho expired events
CONFIG SET notify-keyspace-events "Ex"

# Subscribe để nhận event khi key expire
redis-cli PSUBSCRIBE __keyevent@0__:expired
# → khi session:abc expire, nhận được event "session:abc"
```

Use case: cleanup side effects khi session hết hạn (logout, giải phóng lock).

I UNLINK thay vì DEL cho big keys

```
DEL big_list      # ❗ blocking — nếu list có 1M items, block Redis vài giây
UNLINK big_list   # async delete — trả về ngay, GC chạy background (Redis 4+)
```

9. Performance

Pipelining — giảm round-trip latency

```
# Thay vì 1000 lần SET riêng lẻ (1000 RTT):
redis-cli --pipe << 'EOF'
SET k1 v1
SET k2 v2
SET k3 v3
EOF

# Python redis-py:
pipe = r.pipeline()
for i in range(1000):
    pipe.set(f"key:{i}", f"val:{i}")
pipe.execute()    # 1 RTT thay vì 1000
```

MULTI/EXEC — transaction

```
MULTI
INCR counter
LPUSH recent_actions "buy"
EXPIRE recent_actions 86400
EXEC    # tất cả chạy atomic, không bị interleave

DISCARD    # hủy transaction
```

MULTI/EXEC không rollback khi command lỗi runtime. Chỉ QUEUED phase errors mới cancel. Nếu cần rollback thực sự → dùng Lua.

Slow log — tìm query chậm

```
CONFIG SET slowlog-log-slower-than 10000   # 10ms, đơn vị microseconds
CONFIG SET slowlog-max-len 128

SLOWLOG GET 10     # 10 entries gần nhất
SLOWLOG LEN        # tổng số entries
SLOWLOG RESET
# Output: [id, timestamp, duration_us, [command args], client_addr, client_name]
```

Latency monitoring

```
CONFIG SET latency-monitor-threshold 100   # ms
LATENCY LATEST         # xem event latency gần nhất
LATENCY HISTORY event # lịch sử latency của event
LATENCY RESET          # reset counter

# Intrinsic latency test (baseline OS latency)
redis-cli --intrinsic-latency 60   # đo 60s
```

Big key detection

```
# Scan và report big keys
redis-cli --bigkeys -i 0.1    # -i: sleep 0.1s mỗi 100 keys (production safe)
```

```
# Manual: kiểm tra specific key
DEBUG OBJECT mykey    # → serializedlength, encoding
LLEN mylist           # với list
SCARD myset           # với set
HLEN myhash           # với hash
```

---

10. Security

ACL — user-based access control (Redis 6+)

```
# Tạo user chỉ đọc, chỉ được access key bắt đầu bằng "data:"
ACL SETUSER readonly on >password123 ~data:* +GET +MGET +LRANGE +HGETALL

# Tạo user app với full access trên namespace riêng
ACL SETUSER app_user on >app_secret ~app:* +@all

ACL LIST           # xem tất cả users
ACL WHOAMI         # user hiện tại
ACL LOG            # xem access violations
```

TLS và protected mode

```
# redis.conf
tls-port 6380
tls-cert-file /etc/redis/tls/redis.crt
tls-key-file /etc/redis/tls/redis.key
tls-ca-cert-file /etc/redis/tls/ca.crt

# Bind chỉ localhost + internal IP
bind 127.0.0.1 10.0.0.5

# Protected mode — từ chối kết nối nếu không có password và bind = all
protected-mode yes
requirepass your_strong_password
```

Rename dangerous commands

```
# redis.conf — đổi tên command nguy hiểm
rename-command FLUSHALL ""        # disable hoàn toàn
rename-command CONFIG "CONFIG_ADMIN_ONLY_a3f8b"
rename-command DEBUG ""
rename-command EVAL ""            # nếu không cần Lua
```

Rename trong redis.conf yêu cầu restart. ACL là cách hiện đại hơn để giới hạn command.

---

11. Patterns

Cache-aside pattern

```
# 1. Đọc từ cache
val = redis.get(f"user:{user_id}")
if val:
    return json.loads(val)
```

```
# 2. Cache miss → đọc từ DB
user = db.query("SELECT * FROM users WHERE id = %s", user_id)

# 3. Ghi vào cache với TTL
redis.setex(f"user:{user_id}", 300, json.dumps(user))
return user
```

I Session store

```
# Tạo session với TTL
SET session:{uuid} '{"user_id":1001,"role":"admin"}' EX 3600

# Refresh TTL khi user active (sliding expiry)
EXPIRE session:{uuid} 3600

# Xóa session khi logout
DEL session:{uuid}
```

A Redlock — distributed lock across multiple Redis nodes

```
# Acquire lock trên N=5 nodes (majority = 3)
# Mỗi node: SET lock:resource token NX PX 30000
# Thành công nếu ≥ 3 node accept VÀ elapsed < 30000ms

# Python: pip install redis-py-lock
from redlock import Redlock
dlm = Redlock([{"host": "node1"}, {"host": "node2"}, {"host": "node3"}])
with dlm.lock("lock:order:42", 30000):
    process_order(42)
```

! Redlock có tranh cãi về correctness (Martin Kleppmann critique). Với critical financial ops, xem xét dùng database transaction thay thế.

I Leaderboard với time decay

```
# Score = raw_score / (1 + hours_since_post)^gravity
# Tính ngoài Redis rồi ZADD

# Top 10 global
ZRANGE leaderboard 0 9 REV WITHSCORES

# Top 10 trong khoảng score
ZRANGEBYSCORE leaderboard 1000 +inf REV LIMIT 0 10

# Rank của user
ZREVRANK leaderboard "alice"    # 0-indexed từ top
```

---

12. Debugging

B MONITOR — xem tất cả commands real-time

```
redis-cli MONITOR
# ! Performance impact ~50%. CHỈ dùng debug ngắn hạn, không production
```

CLIENT LIST — active connections

```
CLIENT LIST
# → id=3 addr=127.0.0.1:52341 fd=8 cmd=get age=0 idle=0 flags=N db=0

CLIENT KILL ID 3          # kill connection cụ thể
CLIENT KILL ADDR 1.2.3.4  # kill theo IP
CLIENT SETNAME myapp      # đặt tên connection (debug dễ hơn)
```

INFO sections

```
INFO server      # version, OS, uptime
INFO memory      # used_memory, mem_fragmentation_ratio
INFO stats       # ops/sec, hits, misses, evictions
INFO replication # role, connected_slaves, offset
INFO keyspace    # số key per DB, expires

# Hit rate
INFO stats | grep -E "keyspace_hits|keyspace_misses"
# hit_rate = hits / (hits + misses)
```

DEBUG commands

```
DEBUG SLEEP 2             # block Redis 2s — test timeout behavior
DEBUG JMAP                # dump Java-style memory (nếu build hỗ trợ)
DEBUG SET-ACTIVE-EXPIRE 0 # tắt active expiry (test only)
OBJECT ENCODING key       # xem encoding hiện tại
OBJECT REFCOUNT key       # reference count
OBJECT IDLETIME key       # giây từ lần cuối access
OBJECT FREQ key           # access frequency (chỉ với LFU policy)
OBJECT HELP               # list tất cả OBJECT subcommands
```

redis-cli tricks

```
# Kết nối với password
redis-cli -h host -p 6379 -a password

# Chạy command không interactive
redis-cli PING
redis-cli INFO memory | grep used_memory_human

# Measure latency liên tục
redis-cli --latency -h host

# Stat mode — xem ops/sec realtime
redis-cli --stat

# Scan tất cả key của pattern và xóa (production safe)
redis-cli --scan --pattern "temp:*" | xargs redis-cli DEL

# Dump và restore key giữa 2 Redis
redis-cli --rdb /tmp/dump.rdb
```

Cập nhật: 2026-05-27 | Redis 7.x

08

PostgreSQL Tricks — Beginner to Advanced

> Giải thích bằng tiếng Việt, lệnh bằng tiếng Anh. Tags: B beginner · I intermediate · A advanced · ! gotcha

1. psql Productivity

B Các shortcut hay dùng nhất

```
\l          -- list databases
\c dbname   -- connect sang database khác
\dt         -- list tables trong schema hiện tại
\dt *.*     -- list tất cả tables mọi schema
\d users    -- describe table (columns, indexes, constraints)
\di         -- list indexes
\dv         -- list views
\df         -- list functions
\dn         -- list schemas
\du         -- list roles/users
\?          -- help về psql backslash commands
\h SELECT   -- help về SQL syntax
\q          -- quit
```

B Expanded display và timing

```
\x          -- toggle expanded display (vertical format cho row rộng)
\x auto     -- tự chuyển expanded nếu output quá rộng
\timing     -- bật/tắt hiển thị execution time sau mỗi query
\watch 2    -- chạy lại query trước mỗi 2 giây (realtime monitoring)

-- Ví dụ: monitor số connections mỗi 3s
SELECT count(*), state FROM pg_stat_activity GROUP BY state;
\watch 3
```

I COPY — import/export nhanh

```
-- Export query ra CSV
\COPY (SELECT id, name, email FROM users WHERE active = true)
  TO '/tmp/users.csv' WITH CSV HEADER;

-- Import CSV vào table
\COPY users(name, email, created_at)
  FROM '/tmp/import.csv' WITH CSV HEADER DELIMITER ',';

-- Import với NULL handling
\COPY orders FROM '/tmp/orders.csv'
  WITH CSV HEADER NULL 'NULL' DELIMITER '|';
```

! `\COPY` (backslash) chạy phía client — không cần superuser. `COPY` (không backslash) chạy phía server — cần superuser và path phải accessible bởi postgres process.

I .psqlrc customization

```
# ~/.psqlrc
\set QUIET 1
\set PROMPT1 '%[%033[1;32m%]%n@%/%[%033[0m%]%R%# '
\set PROMPT2 '%[%033[1;33m%]...%[%033[0m%] '
\timing on
\x auto
\set HISTSIZE 10000
\set HISTFILE ~/.psql_history
\set COMP_KEYWORD_CASE upper
-- Tắt pager cho output nhỏ
\pset pager off
\set QUIET 0
```

I Pager và format output

```
\pset format aligned     -- default
\pset format csv         -- CSV output
\pset format wrapped     -- wrap long lines
\pset pager always       -- luôn dùng pager (less)
\pset pager off          -- tắt pager

-- Export kết quả sang file
\o /tmp/output.txt
SELECT * FROM big_table;
\o    -- tắt redirect
```

---

2. Query Optimization

B EXPLAIN ANALYZE — đọc query plan

```
EXPLAIN SELECT * FROM orders WHERE user_id = 1001;
-- → chỉ show plan, KHÔNG chạy query

EXPLAIN ANALYZE SELECT * FROM orders WHERE user_id = 1001;
-- → chạy query thật, show actual rows vs estimated

EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT)
  SELECT * FROM orders WHERE user_id = 1001;
-- BUFFERS: show cache hit/miss — quan trọng để debug I/O
```

Đọc plan: tìm node có `actual rows` >> `rows` (estimated) → bad statistics → chạy `ANALYZE table`.

I Index types — khi nào dùng gì

```
-- B-tree (default) — equality, range, ORDER BY, LIKE 'prefix%'
CREATE INDEX idx_orders_user ON orders(user_id);
CREATE INDEX idx_orders_created ON orders(created_at DESC);

-- GIN — full-text search, JSONB, array containment
CREATE INDEX idx_products_tags ON products USING GIN(tags);
CREATE INDEX idx_docs_body ON documents USING GIN(to_tsvector('english', body));
```

```
-- GiST — geometric types, IP ranges, fuzzy string (pg_trgm)
CREATE INDEX idx_locations_geo ON locations USING GIST(coordinates);

-- BRIN — rất nhỏ, chỉ hiệu quả với data có natural order (time-series, log tables)
CREATE INDEX idx_events_ts ON events USING BRIN(occurred_at) WITH (pages_per_range = 128);
-- BRIN tốt khi table hàng tỷ rows và data insert tuần tự
```

Partial index — index chỉ một phần data

```
-- Chỉ index active users (giả sử 90% là inactive)
CREATE INDEX idx_users_active_email ON users(email)
  WHERE active = true;

-- Chỉ index orders chưa xử lý
CREATE INDEX idx_orders_pending ON orders(created_at)
  WHERE status = 'pending';

-- Query PHẢI có điều kiện khớp WHERE clause mới dùng được index này
SELECT * FROM users WHERE email = 'x@y.com' AND active = true;
```

Covering index (INCLUDE)

```
-- Index-only scan: PostgreSQL không cần đọc heap nếu tất cả columns cần đều trong index
CREATE INDEX idx_orders_covering ON orders(user_id)
  INCLUDE (status, total_amount, created_at);

-- Query này sẽ dùng index-only scan (không đọc table):
SELECT status, total_amount, created_at
FROM orders
WHERE user_id = 1001;
```

REINDEX CONCURRENTLY — rebuild index không lock

```
-- Tránh ACCESS EXCLUSIVE lock khi reindex
REINDEX INDEX CONCURRENTLY idx_orders_user;
REINDEX TABLE CONCURRENTLY orders;    -- rebuild tất cả index của table

-- Check index bloat trước khi reindex
SELECT schemaname, tablename, indexname,
       pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
ORDER BY pg_relation_size(indexrelid) DESC;
```

`REINDEX CONCURRENTLY` cần nhiều disk space hơn (build index mới song song index cũ). Có thể fail nếu hết disk.

---

pg_stat_user_indexes — tìm unused indexes

```
SELECT
  schemaname,
  tablename,
  indexname,
  idx_scan,     -- số lần index được dùng
  pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE idx_scan = 0        -- chưa từng được dùng
```

```
  AND schemaname = 'public'
ORDER BY pg_relation_size(indexrelid) DESC;
```

3. Advanced Indexing

A Expression index — index trên biểu thức

```
-- Index trên lower(email) để tìm kiếm case-insensitive
CREATE INDEX idx_users_email_lower ON users(lower(email));

-- Query phải dùng CÙNG biểu thức
SELECT * FROM users WHERE lower(email) = lower('User@Example.com');

-- Index trên extracted JSON field
CREATE INDEX idx_orders_metadata_source
  ON orders((metadata->>'source'));

-- Index trên date part
CREATE INDEX idx_events_month ON events(date_trunc('month', occurred_at));
```

I Multicolumn index — thứ tự cột quan trọng

```
-- Index (a, b) có thể dùng cho: WHERE a=?, WHERE a=? AND b=?
-- KHÔNG hiệu quả cho: WHERE b=? (chỉ b)
CREATE INDEX idx_orders_user_status ON orders(user_id, status);

-- Thứ tự tốt: cột có selectivity cao hơn → đứng trước
-- Ngoại lệ: nếu query hay có a=? AND b=? thì thứ tự ít quan trọng hơn
SELECT * FROM orders WHERE user_id = 1001 AND status = 'pending';  -- dùng index tốt
SELECT * FROM orders WHERE status = 'pending';                     -- sequential scan
```

4. Window Functions

I ROW_NUMBER, RANK, DENSE_RANK

```
-- ROW_NUMBER: số thứ tự duy nhất, không có tie
-- RANK: tie → cùng rank, số tiếp theo bị skip
-- DENSE_RANK: tie → cùng rank, không skip

SELECT
  user_id,
  score,
  ROW_NUMBER() OVER (ORDER BY score DESC) AS row_num,
  RANK()       OVER (ORDER BY score DESC) AS rank,
  DENSE_RANK() OVER (ORDER BY score DESC) AS dense_rank
FROM leaderboard;

-- Partition: rank riêng theo từng category
SELECT
  product_id,
  category,
  revenue,
  RANK() OVER (PARTITION BY category ORDER BY revenue DESC) AS rank_in_category
FROM sales;
```

I LAG/LEAD — so sánh với row trước/sau

```sql
-- MoM growth: so sánh revenue tháng này với tháng trước
SELECT
  month,
  revenue,
  LAG(revenue, 1) OVER (ORDER BY month) AS prev_month_revenue,
  revenue - LAG(revenue, 1) OVER (ORDER BY month) AS diff,
  ROUND(
    100.0 * (revenue - LAG(revenue, 1) OVER (ORDER BY month))
    / NULLIF(LAG(revenue, 1) OVER (ORDER BY month), 0),
    2
  ) AS growth_pct
FROM monthly_revenue;
```

I Running totals và percentiles

```sql
-- Running total (cumulative sum)
SELECT
  order_date,
  amount,
  SUM(amount) OVER (ORDER BY order_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS runn
ing_total
FROM orders;

-- 7-day moving average
SELECT
  date,
  value,
  AVG(value) OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS ma7
FROM daily_metrics;

-- Percentile
SELECT
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY response_time_ms) AS p50,
  PERCENTILE_CONT(0.95) WITHIN GROUP (ORDER BY response_time_ms) AS p95,
  PERCENTILE_CONT(0.99) WITHIN GROUP (ORDER BY response_time_ms) AS p99
FROM api_logs;
```

A Gaps and Islands

```sql
-- Tìm khoảng liên tiếp (islands) trong sequential data
WITH numbered AS (
  SELECT
    user_id,
    login_date,
    login_date - (ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY login_date))::int AS grp
  FROM user_logins
)
SELECT
  user_id,
  MIN(login_date) AS streak_start,
  MAX(login_date) AS streak_end,
  COUNT(*) AS streak_length
FROM numbered
GROUP BY user_id, grp
ORDER BY user_id, streak_start;
```

---

5. CTEs & Recursive Queries

I Materialized CTE — kiểm soát optimization fence

```
-- Mặc định PG 12+: CTE có thể bị inlined (optimizer quyết định)
-- MATERIALIZED: force CTE chạy một lần, kết quả được cache
WITH expensive_calc AS MATERIALIZED (
  SELECT user_id, SUM(amount) AS total
  FROM orders
  GROUP BY user_id
)
SELECT u.name, ec.total
FROM users u
JOIN expensive_calc ec ON u.id = ec.user_id
WHERE ec.total > 1000;

-- NOT MATERIALIZED: force inline (hint cho optimizer)
WITH recent AS NOT MATERIALIZED (
  SELECT * FROM events WHERE occurred_at > NOW() - INTERVAL '1 hour'
)
SELECT * FROM recent WHERE type = 'error';
```

I Recursive CTE — hierarchy traversal

```
-- Lấy toàn bộ cây tổ chức bên dưới manager_id = 5
WITH RECURSIVE org_tree AS (
  -- Base case: node gốc
  SELECT id, name, manager_id, 0 AS depth
  FROM employees
  WHERE id = 5

  UNION ALL

  -- Recursive case: tìm direct reports
  SELECT e.id, e.name, e.manager_id, ot.depth + 1
  FROM employees e
  JOIN org_tree ot ON e.manager_id = ot.id
)
SELECT * FROM org_tree ORDER BY depth, name;
```

A Cycle detection trong recursive CTE

```
-- PostgreSQL 14+: CYCLE clause tự động detect cycle
WITH RECURSIVE graph_traverse AS (
  SELECT node_id, neighbor_id, ARRAY[node_id] AS path
  FROM edges
  WHERE node_id = 1

  UNION ALL

  SELECT e.node_id, e.neighbor_id, path || e.node_id
  FROM edges e
  JOIN graph_traverse gt ON e.node_id = gt.neighbor_id
  WHERE e.node_id <> ALL(path)    -- manual cycle detection (PG 13 trở xuống)
)
CYCLE node_id SET is_cycle USING cycle_path    -- PG 14+ syntax
SELECT * FROM graph_traverse WHERE NOT is_cycle;
```

6. JSON/JSONB

B Operators cơ bản

```sql
-- JSONB operators
SELECT data→'address'          FROM users;    -- → JSON object
SELECT data→>'address'         FROM users;    -- → text (unquoted)
SELECT data→'address'→>'city' FROM users;    -- → nested field as text
SELECT data#>>'{address,city}'  FROM users;    -- path syntax

-- Containment: tìm rows có JSON chứa subset
SELECT * FROM products WHERE metadata @> '{"brand": "Nike"}';

-- Key exists
SELECT * FROM users WHERE data ? 'phone_number';

-- Any key exists
SELECT * FROM users WHERE data ?| ARRAY['phone', 'mobile'];
```

I Indexing JSONB

```sql
-- GIN index cho toàn bộ JSONB column
CREATE INDEX idx_products_metadata ON products USING GIN(metadata);

-- GIN index cho specific path (nhỏ hơn, nhanh hơn)
CREATE INDEX idx_products_brand
  ON products USING GIN((metadata→'brand'));

-- Expression index cho equality lookup
CREATE INDEX idx_users_country
  ON users((profile→>'country'));

SELECT * FROM users WHERE profile→>'country' = 'VN';    -- dùng index
```

A jsonb_path_query — JSONPath syntax (PG 12+)

```sql
-- Tìm tất cả products có price > 100 trong nested array
SELECT jsonb_path_query(
  '{"items": [{"name":"A","price":50},{"name":"B","price":150}]}',
  '$.items[*] ? (@.price > 100)'
);
-- → {"name": "B", "price": 150}

-- Filter với jsonb_path_exists
SELECT * FROM orders
WHERE jsonb_path_exists(line_items, '$.* ? (@.qty > 10)');
```

I Generated columns từ JSON

```sql
-- Tạo column được tính tự động từ JSONB (immutable)
ALTER TABLE users
  ADD COLUMN country TEXT GENERATED ALWAYS AS (profile→>'country') STORED;

-- Giờ có thể index và query bình thường
CREATE INDEX idx_users_country ON users(country);
SELECT * FROM users WHERE country = 'VN';
```

7. Partitioning

I Range partitioning — time-series data

```
-- Parent table
CREATE TABLE events (
  id BIGSERIAL,
  occurred_at TIMESTAMPTZ NOT NULL,
  type TEXT,
  payload JSONB
) PARTITION BY RANGE (occurred_at);

-- Tạo partitions theo tháng
CREATE TABLE events_2026_05 PARTITION OF events
  FOR VALUES FROM ('2026-05-01') TO ('2026-06-01');

CREATE TABLE events_2026_06 PARTITION OF events
  FOR VALUES FROM ('2026-06-01') TO ('2026-07-01');

-- Default partition: chứa rows không khớp partition nào
CREATE TABLE events_default PARTITION OF events DEFAULT;
```

I Partition pruning — verify planner bỏ qua đúng partitions

```
EXPLAIN SELECT * FROM events
WHERE occurred_at BETWEEN '2026-05-01' AND '2026-05-31';
-- → planner chỉ scan events_2026_05, bỏ qua các partition khác
-- Tìm "Partitions selected" trong plan
```

A Attach/Detach partition — zero-downtime archival

```
-- Detach partition cũ (nhanh, gần như instant)
ALTER TABLE events DETACH PARTITION events_2024_01 CONCURRENTLY;
-- PG 14+: CONCURRENTLY tránh lock lâu

-- Partition cũ giờ là standalone table → backup → drop
pg_dump -t events_2024_01 mydb > events_2024_01_archive.sql
DROP TABLE events_2024_01;

-- Attach table mới (phải có check constraint trước)
ALTER TABLE new_events_2026_07
  ADD CONSTRAINT chk_date CHECK (occurred_at ⩾ '2026-07-01' AND occurred_at < '2026-08-01');
ALTER TABLE events ATTACH PARTITION new_events_2026_07
  FOR VALUES FROM ('2026-07-01') TO ('2026-08-01');
```

8. Replication

I Streaming replication — setup cơ bản

```
-- Trên primary: tạo replication user
CREATE USER replicator WITH REPLICATION LOGIN PASSWORD 'rep_password';

-- pg_hba.conf trên primary
-- host replication replicator replica-ip/32 md5

-- pg_basebackup để clone primary sang replica
pg_basebackup -h primary-host -U replicator -D /var/lib/postgresql/data \
```

```
  -Fp -Xs -P -R
-- -R: tự tạo standby.signal và primary_conninfo
```

A Logical replication — table-level, cross-version

```
-- Trên source DB: tạo publication
CREATE PUBLICATION my_pub FOR TABLE users, orders;
-- Hoặc tất cả tables:
CREATE PUBLICATION my_pub FOR ALL TABLES;

-- Trên target DB: tạo subscription
CREATE SUBSCRIPTION my_sub
  CONNECTION 'host=source-host dbname=mydb user=rep password=xxx'
  PUBLICATION my_pub;

-- Monitor replication lag
SELECT
  subname,
  received_lsn,
  latest_end_lsn,
  latest_end_time
FROM pg_stat_subscription;
```

! Logical replication không replicate DDL (ALTER TABLE, CREATE INDEX...). Schema phải được sync thủ công.

9. Backup & Recovery

B pg_dump tricks

```
# Dump single table
pg_dump -t users mydb > users.sql

# Dump với compression
pg_dump -Fc mydb > mydb.dump          # custom format, ~5x nhỏ hơn

# Dump chỉ schema (không data)
pg_dump -s mydb > schema_only.sql

# Dump chỉ data (không schema)
pg_dump -a mydb > data_only.sql

# Exclude table
pg_dump --exclude-table=audit_logs mydb > mydb_no_audit.sql
```

I pg_restore parallel — restore nhanh hơn

```
# Restore với 4 parallel jobs (chỉ với custom format -Fc)
pg_restore -Fc -j 4 -d mydb mydb.dump

# Restore chỉ specific table
pg_restore -Fc -t orders -d mydb mydb.dump

# List nội dung dump trước khi restore
pg_restore -l mydb.dump | head -50
```

A PITR với WAL archiving

```
# postgresql.conf
archive_mode = on
archive_command = 'cp %p /mnt/wal-archive/%f'
# Hoặc dùng pgBackRest/Barman cho production

# Recovery target (restore đến time cụ thể)
# recovery.conf (PG 12 trở xuống) hoặc postgresql.conf (PG 12+)
restore_command = 'cp /mnt/wal-archive/%f %p'
recovery_target_time = '2026-05-27 14:30:00+07'
recovery_target_action = 'promote'
```

---

10. Locks & Concurrency

I Advisory locks — application-level locking

```
-- Session-level advisory lock (tự động release khi session kết thúc)
SELECT pg_try_advisory_lock(12345);    -- → true nếu acquire được
SELECT pg_advisory_unlock(12345);

-- Transaction-level (tự động release khi transaction kết thúc)
SELECT pg_try_advisory_xact_lock(12345);

-- Dùng để prevent concurrent job execution
DO $$
BEGIN
  IF pg_try_advisory_lock(hashtext('cron:cleanup_job')) THEN
    -- chạy job
    PERFORM cleanup_old_records();
    PERFORM pg_advisory_unlock(hashtext('cron:cleanup_job'));
  END IF;
END $$;
```

I Lock monitoring

```
-- Xem locks đang chờ
SELECT
  pid,
  wait_event_type,
  wait_event,
  query,
  state,
  pg_blocking_pids(pid) AS blocked_by
FROM pg_stat_activity
WHERE wait_event_type = 'Lock';

-- Xem chi tiết lock graph
SELECT
  blocked.pid AS blocked_pid,
  blocked.query AS blocked_query,
  blocking.pid AS blocking_pid,
  blocking.query AS blocking_query
FROM pg_stat_activity blocked
JOIN pg_stat_activity blocking
  ON blocking.pid = ANY(pg_blocking_pids(blocked.pid));
```

I SKIP LOCKED và NOWAIT — non-blocking queue

```
-- Worker pattern: lấy job mà không chờ lock của worker khác
BEGIN;
SELECT id, payload
FROM jobs
WHERE status = 'pending'
ORDER BY created_at
LIMIT 1
FOR UPDATE SKIP LOCKED;   -- bỏ qua row đang bị lock bởi worker khác
-- process job...
UPDATE jobs SET status = 'done' WHERE id = :job_id;
COMMIT;

-- NOWAIT: fail ngay nếu không lấy được lock (không chờ)
SELECT * FROM orders WHERE id = 42 FOR UPDATE NOWAIT;
-- → ERROR nếu row đang bị lock
```

---

11. Performance

I Tuning key parameters

```
-- Kiểm tra current settings
SHOW shared_buffers;     -- nên = 25% RAM (mặc định 128MB quá thấp)
SHOW work_mem;           -- RAM per sort/hash operation (cẩn thận: per-query)
SHOW effective_cache_size;  -- hint cho planner về OS cache

-- Cập nhật trong postgresql.conf hoặc:
ALTER SYSTEM SET shared_buffers = '4GB';
ALTER SYSTEM SET work_mem = '64MB';
ALTER SYSTEM SET effective_cache_size = '12GB';
SELECT pg_reload_conf();  -- reload (shared_buffers cần restart)
```

! `work_mem` là per operation, không per connection. Nếu query có 5 sort steps và 100 connections → có thể dùng 5 × 100 × work_mem RAM.

I pg_stat_statements — top slow queries

```
-- Cài extension (cần superuser)
CREATE EXTENSION pg_stat_statements;

-- Top 10 queries tốn thời gian nhất
SELECT
  round(total_exec_time::numeric, 2) AS total_ms,
  calls,
  round(mean_exec_time::numeric, 2) AS mean_ms,
  round(stddev_exec_time::numeric, 2) AS stddev_ms,
  left(query, 100) AS query_snippet
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;

-- Reset stats
SELECT pg_stat_statements_reset();
```

A auto_explain — log slow query plans tự động

```
-- postgresql.conf
shared_preload_libraries = 'auto_explain'
```

```
auto_explain.log_min_duration = 1000   -- log plan nếu query > 1s
auto_explain.log_analyze = true        -- bao gồm actual rows
auto_explain.log_buffers = true
auto_explain.log_nested_statements = true

-- Reload
SELECT pg_reload_conf();
```

A VACUUM tuning

```
-- Xem bloat và vacuum stats
SELECT
  relname,
  n_dead_tup,
  n_live_tup,
  round(n_dead_tup::numeric / NULLIF(n_live_tup + n_dead_tup, 0) * 100, 2) AS dead_pct,
  last_autovacuum,
  last_autoanalyze
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC;

-- Manual vacuum
VACUUM ANALYZE orders;        -- vacuum + update statistics
VACUUM FULL orders;           -- ❗ lock table, reclaim disk — dùng pg_repack thay thế

-- Tune autovacuum per-table cho high-write tables
ALTER TABLE orders SET (
  autovacuum_vacuum_scale_factor = 0.01,   -- trigger khi 1% rows là dead (mặc định 20%)
  autovacuum_analyze_scale_factor = 0.005
);
```

12. Security

I Row-Level Security (RLS)

```
-- Bật RLS trên table
ALTER TABLE documents ENABLE ROW LEVEL SECURITY;

-- Policy: user chỉ thấy documents của mình
CREATE POLICY user_isolation ON documents
  USING (owner_id = current_setting('app.current_user_id')::bigint);

-- Policy riêng cho SELECT và INSERT
CREATE POLICY select_own ON orders FOR SELECT
  USING (user_id = current_setting('app.user_id')::int);

CREATE POLICY insert_own ON orders FOR INSERT
  WITH CHECK (user_id = current_setting('app.user_id')::int);

-- Set context trong application
SET app.current_user_id = '1001';

-- Bypass RLS (superuser hoặc BYPASSRLS role)
ALTER USER app_admin BYPASSRLS;
```

I Column-level grants

```
-- Grant chỉ một số columns
GRANT SELECT (id, name, email) ON users TO readonly_role;
GRANT UPDATE (email, phone) ON users TO support_role;
```

```
-- Revoke
REVOKE ALL ON users FROM public;
```

A pg_hba.conf patterns

```
# TYPE  DATABASE  USER      ADDRESS       METHOD
local   all       postgres                peer        # local unix socket
host    mydb      app_user  10.0.0.0/8    scram-sha-256
host    all       all       0.0.0.0/0     reject      # deny tất cả còn lại

# Reload sau khi thay đổi
SELECT pg_reload_conf();
```

A Audit với pgaudit

```
-- postgresql.conf
shared_preload_libraries = 'pgaudit'
pgaudit.log = 'write, ddl'    -- log INSERT/UPDATE/DELETE/TRUNCATE và DDL
pgaudit.log_relation = on     -- bao gồm tên table

-- Per-role audit
ALTER ROLE sensitive_user SET pgaudit.log = 'all';
```

13. Useful Extensions

I pg_trgm — fuzzy search và LIKE bất kỳ vị trí

```
CREATE EXTENSION pg_trgm;

-- Similarity search
SELECT name, similarity(name, 'nguyen van a') AS sim
FROM users
WHERE similarity(name, 'nguyen van a') > 0.3
ORDER BY sim DESC;

-- GIN/GiST index cho LIKE '%keyword%' (thường không dùng được B-tree)
CREATE INDEX idx_users_name_trgm ON users USING GIN(name gin_trgm_ops);
SELECT * FROM users WHERE name ILIKE '%hieu%';   -- dùng index
```

I pgcrypto — mã hóa trong database

```
CREATE EXTENSION pgcrypto;

-- Hash password
INSERT INTO users(email, password_hash)
VALUES ('user@example.com', crypt('my_password', gen_salt('bf', 12)));

-- Verify
SELECT * FROM users
WHERE email = 'user@example.com'
  AND password_hash = crypt('my_password', password_hash);

-- Encrypt/decrypt data
SELECT pgp_sym_encrypt('sensitive data', 'secret_key');
SELECT pgp_sym_decrypt(encrypted_col::bytea, 'secret_key') FROM secrets;
```

pg_cron — cron jobs trong PostgreSQL

```
CREATE EXTENSION pg_cron;

-- Chạy cleanup mỗi ngày lúc 3am
SELECT cron.schedule('cleanup-old-events', '0 3 * * *',
  $$DELETE FROM events WHERE occurred_at < NOW() - INTERVAL '90 days'$$);

-- List jobs
SELECT * FROM cron.job;

-- Xóa job
SELECT cron.unschedule('cleanup-old-events');

-- Xem lịch sử chạy
SELECT * FROM cron.job_run_details ORDER BY start_time DESC LIMIT 20;
```

TimescaleDB basics — time-series trên PostgreSQL

```
CREATE EXTENSION timescaledb;

-- Convert table thành hypertable (partitioned by time tự động)
SELECT create_hypertable('metrics', 'time', chunk_time_interval ⇒ INTERVAL '1 week');

-- Continuous aggregate — materialized view tự động refresh
CREATE MATERIALIZED VIEW metrics_hourly
WITH (timescaledb.continuous) AS
SELECT
  time_bucket('1 hour', time) AS bucket,
  AVG(value) AS avg_value,
  MAX(value) AS max_value
FROM metrics
GROUP BY bucket;

-- Data retention policy — tự động drop chunks cũ
SELECT add_retention_policy('metrics', INTERVAL '90 days');

-- Compression policy — compress chunks > 7 ngày
SELECT add_compression_policy('metrics', INTERVAL '7 days');
```

TimescaleDB hypertable: một số PostgreSQL features bị hạn chế (foreign keys pointing to hypertable, UPDATE/DELETE có thể chậm hơn nếu không có time filter).

PostGIS basics — geospatial queries

```
CREATE EXTENSION postgis;

-- Thêm geometry column
ALTER TABLE locations ADD COLUMN coordinates GEOMETRY(POINT, 4326);

-- Insert điểm (longitude, latitude)
UPDATE locations
SET coordinates = ST_SetSRID(ST_MakePoint(106.6297, 10.8231), 4326)
WHERE id = 1;    -- HCM City

-- Tìm các địa điểm trong bán kính 5km
SELECT name, ST_Distance(
  coordinates::geography,
  ST_SetSRID(ST_MakePoint(106.6297, 10.8231), 4326)::geography
) AS distance_meters
FROM locations
WHERE ST_DWithin(
```

```
  coordinates::geography,
  ST_SetSRID(ST_MakePoint(106.6297, 10.8231), 4326)::geography,
  5000    -- 5000 meters
)
ORDER BY distance_meters;

-- Spatial index
CREATE INDEX idx_locations_geo ON locations USING GIST(coordinates);
```

Cập nhật: 2026-05-27 | PostgreSQL 16.x

09

Terraform Tricks — Beginner to Advanced

> Giải thích bằng tiếng Việt. Commands/config bằng tiếng Anh. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

CLI Productivity

B Format & Validate trước khi làm gì

Luôn chạy `fmt` và `validate` trước khi plan. `fmt` sửa indent/style, `validate` bắt lỗi syntax/type mà không cần gọi provider API.

```
terraform fmt -recursive        # format toàn bộ thư mục con
terraform validate              # kiểm tra cấu trúc + type, không gọi API
```

B Lưu plan ra file để apply chính xác

`plan -out` lưu execution plan, `apply` đọc plan đó — đảm bảo apply đúng những gì đã review, không bị thay đổi giữa chừng.

```
terraform plan -out=tfplan.bin
terraform apply tfplan.bin
```

! File `.bin` chứa state snapshot, có thể chứa secrets — đừng commit vào git.

I Targeted apply — chỉ deploy một resource

Dùng khi cần apply nhanh một resource cụ thể mà không ảnh hưởng phần còn lại. Hữu ích khi debug.

```
terraform apply -target=aws_instance.web
terraform apply -target=module.vpc -target=aws_security_group.allow_ssh
```

! `-target` bỏ qua dependency graph — dùng xong phải full apply để sync lại.

I Refresh-only plan — phát hiện drift

Chỉ refresh state từ cloud, không thay đổi gì. Dùng để kiểm tra xem có ai thay đổi infra ngoài Terraform không.

```
terraform plan -refresh-only
terraform apply -refresh-only   # cập nhật state file khớp với thực tế cloud
```

I Terraform console — debug expression

Console là REPL để test expression, function, variable trước khi dùng trong code.

```
terraform console

> cidrsubnet("10.0.0.0/16", 8, 1)
```

```
"10.0.1.0/24"
> length(var.availability_zones)
3
> toset(["a", "b", "a"])
toset(["a", "b"])
```

A Visualize dependency graph

Export graph sang DOT format để xem dependency giữa các resource.

```
terraform graph | dot -Tsvg > graph.svg
terraform graph -type=plan | dot -Tpng > plan-graph.png
```

State Management

B Xem và filter state

Liệt kê resource trong state, xem chi tiết một resource.

```
terraform state list                          # danh sách tất cả
terraform state list aws_instance.*           # filter theo pattern
terraform state show aws_instance.web         # xem chi tiết JSON
```

I Di chuyển resource trong state

Khi rename resource hoặc di chuyển vào module mà không muốn destroy/recreate.

```
# rename resource
terraform state mv aws_instance.old aws_instance.new

# di chuyển vào module
terraform state mv aws_instance.web module.compute.aws_instance.web
```

I Xóa resource khỏi state mà không destroy

Dùng khi muốn Terraform "quên" resource — resource vẫn tồn tại trên cloud.

```
terraform state rm aws_instance.legacy
terraform state rm module.old_module
```

! Sau khi rm, nếu resource vẫn trong config thì plan sẽ muốn create lại — phải xóa config hoặc import.

I Pull/push state thủ công

Dùng khi cần debug state hoặc migrate giữa backends.

```
terraform state pull > backup.tfstate         # download state hiện tại
terraform state push backup.tfstate           # upload state (nguy hiểm!)
```

! `push` override state hiện tại — dùng cẩn thận, có thể gây conflict.

I State locking

Remote backends (S3+DynamoDB, GCS, Consul) tự động lock state khi có operation đang chạy.

```
# S3 backend với DynamoDB locking
terraform {
  backend "s3" {
    bucket         = "my-terraform-state"
```

```
    key            = "prod/terraform.tfstate"
    region         = "ap-southeast-1"
    dynamodb_table = "terraform-state-lock"
    encrypt        = true
  }
}
```

```
# Nếu state bị stuck lock (process crash):
terraform force-unlock <LOCK_ID>
```

⚠ Chỉ force-unlock khi chắc chắn không có operation nào đang chạy.

A Import resource đã tồn tại

Đưa resource đang tồn tại trên cloud vào quản lý Terraform.

```
# Cách cũ (TF < 1.5)
terraform import aws_s3_bucket.my_bucket my-existing-bucket-name

# Cách mới: import block trong TF 1.5+
```

```
import {
  to = aws_s3_bucket.my_bucket
  id = "my-existing-bucket-name"
}

resource "aws_s3_bucket" "my_bucket" {
  bucket = "my-existing-bucket-name"
}
```

```
terraform plan    # preview import, không destroy/create
terraform apply   # import vào state
```

A Moved blocks — rename không recreate

Khi refactor code (rename resource, chuyển vào module), dùng `moved` block thay vì `state mv`.

```
moved {
  from = aws_instance.app
  to   = module.compute.aws_instance.app
}
```

Lợi thế: declarative, review được qua PR, tự động apply khi plan.

Modules

B Cấu trúc module chuẩn

```
modules/
└── vpc/
    ├── main.tf       # resource definitions
    ├── variables.tf  # input variables
    ├── outputs.tf    # output values
    ├── versions.tf   # required_providers + terraform version
    └── README.md
```

Validate input với validation block

```
variable "environment" {
  type        = string
  description = "Deployment environment"

  validation {
    condition     = contains(["dev", "staging", "prod"], var.environment)
    error_message = "environment phải là: dev, staging, hoặc prod."
  }
}

variable "instance_count" {
  type = number
  validation {
    condition     = var.instance_count > 0 && var.instance_count <= 100
    error_message = "instance_count phải từ 1 đến 100."
  }
}
```

Output patterns — expose đúng thứ cần thiết

```
# outputs.tf trong module
output "vpc_id" {
  description = "ID của VPC"
  value       = aws_vpc.main.id
}

output "private_subnet_ids" {
  description = "Danh sách private subnet IDs"
  value       = aws_subnet.private[*].id
}

output "database_password" {
  description = "Password database (sensitive)"
  value       = random_password.db.result
  sensitive   = true
}
```

Versioning module từ Git

```
module "vpc" {
  source  = "git::https://github.com/org/terraform-modules.git//vpc?ref=v1.2.0"

  cidr_block = "10.0.0.0/16"
}
```

Module composition — module gọi module

```
# Root module
module "network" {
  source      = "./modules/network"
  environment = var.environment
}

module "compute" {
  source     = "./modules/compute"
  vpc_id     = module.network.vpc_id
  subnet_ids = module.network.private_subnet_ids
}
```

Circular dependency giữa modules sẽ báo lỗi — thiết kế data flow một chiều.

Workspaces

B Workspace cơ bản

```
terraform workspace list
terraform workspace new staging
terraform workspace select prod
terraform workspace show          # workspace hiện tại
```

I Dùng workspace trong config

```
locals {
  env_config = {
    dev     = { instance_type = "t3.micro",  min_size = 1 }
    staging = { instance_type = "t3.small",  min_size = 2 }
    prod    = { instance_type = "t3.medium", min_size = 3 }
  }

  config = local.env_config[terraform.workspace]
}

resource "aws_autoscaling_group" "app" {
  min_size      = local.config.min_size
  instance_type = local.config.instance_type
}
```

I Workspace vs separate state files

Workspace phù hợp khi infra structure giống nhau, chỉ khác config. Separate state files phù hợp khi các env có cấu trúc khác nhau hoặc cần isolate hoàn toàn.

```
# Separate state (recommended cho large teams)
environments/
├── dev/
│   ├── main.tf
│   └── terraform.tfvars
├── staging/
│   └── ...
└── prod/
    └── ...
```

! Workspace dùng cùng backend — một lỗi có thể ảnh hưởng nhiều env. Prod nên dùng separate state.

Variables & Locals

B tfvars per environment

```
# File structure
├── variables.tf        # khai báo variables
├── terraform.tfvars    # default values (không commit secret)
├── dev.tfvars
├── staging.tfvars
└── prod.tfvars
```

```
terraform plan -var-file=prod.tfvars
```

I Complex variable types

```
variable "tags" {
  type = map(string)
  default = {
    Project   = "myapp"
    ManagedBy = "terraform"
  }
}

variable "allowed_ips" {
  type    = list(string)
  default = ["10.0.0.0/8"]
}

variable "services" {
  type = map(object({
    port     = number
    protocol = string
    healthy_threshold = optional(number, 2)
  }))
}
```

I Locals để DRY

```
locals {
  name_prefix = "${var.project}-${var.environment}"
  common_tags = merge(var.tags, {
    Environment = var.environment
    Terraform   = "true"
  })

  # computed values
  azs         = slice(data.aws_availability_zones.available.names, 0, 3)
  private_cidrs = [for i, az in local.azs : cidrsubnet(var.vpc_cidr, 8, i)]
}
```

A Sensitive variables — không leak ra log

```
variable "db_password" {
  type      = string
  sensitive = true
}

output "connection_string" {
  value     = "postgres://user:${var.db_password}@${aws_db_instance.main.endpoint}/db"
  sensitive = true   # bắt buộc nếu dùng sensitive variable
}
```

! `sensitive = true` ẩn giá trị trong log nhưng vẫn stored plaintext trong state file.

Data Sources

B Data vs Resource

`resource` tạo/quản lý infrastructure. `data` chỉ đọc thông tin đã tồn tại.

```
# Đọc AMI mới nhất từ AWS
data "aws_ami" "ubuntu" {
  most_recent = true
  owners      = ["099720109477"]  # Canonical
```

```
  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-*-22.04-amd64-server-*"]
  }
}

resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"
}
```

depends_on với data source

Data source thường được resolve trước khi resource tạo — dùng `depends_on` khi data cần resource tồn tại trước.

```
data "aws_lb" "app" {
  name       = "my-app-lb"
  depends_on = [aws_lb.app]   # đảm bảo LB được tạo trước khi query
}
```

terraform_remote_state — đọc output từ state khác

```
data "terraform_remote_state" "network" {
  backend = "s3"
  config = {
    bucket = "my-terraform-state"
    key    = "network/terraform.tfstate"
    region = "ap-southeast-1"
  }
}

resource "aws_instance" "app" {
  subnet_id = data.terraform_remote_state.network.outputs.private_subnet_ids[0]
}
```

External data source — gọi script bên ngoài

```
data "external" "git_info" {
  program = ["bash", "-c", "echo '{\"sha\":\"'$(git rev-parse --short HEAD)'\"}'"]
}

resource "aws_ssm_parameter" "version" {
  name  = "/app/version"
  value = data.external.git_info.result.sha
}
```

External data source không idempotent — chạy mỗi lần plan. Dùng sparingly.

Provisioners & Null Resources

Khi nào dùng provisioner

Provisioner là last resort — dùng khi không có Terraform resource nào phù hợp. Ưu tiên user_data, cloud-init, hoặc config management tools thay thế.

```
resource "aws_instance" "web" {
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t3.micro"

  provisioner "local-exec" {
```

```
    command = "echo ${self.public_ip} >> inventory.txt"
  }
}
```

Null resource với triggers để re-provision

```
resource "null_resource" "ansible_provision" {
  triggers = {
    instance_id     = aws_instance.web.id
    playbook_hash   = filemd5("${path.module}/playbook.yml")
  }

  provisioner "local-exec" {
    command = "ansible-playbook -i ${aws_instance.web.public_ip}, playbook.yml"
  }
}
```

Thay đổi `triggers` sẽ force re-run provisioner.

Dynamic Blocks & For Expressions

Dynamic blocks — tránh lặp code

```
variable "ingress_rules" {
  type = list(object({
    port        = number
    protocol    = string
    cidr_blocks = list(string)
  }))
}

resource "aws_security_group" "app" {
  name = "app-sg"

  dynamic "ingress" {
    for_each = var.ingress_rules
    content {
      from_port   = ingress.value.port
      to_port     = ingress.value.port
      protocol    = ingress.value.protocol
      cidr_blocks = ingress.value.cidr_blocks
    }
  }
}
```

for_each vs count

`count` dùng cho số lượng đơn giản. `for_each` dùng cho map/set — không bị xáo trộn index khi thêm/xóa.

```
# Dùng for_each với map — index ổn định
resource "aws_iam_user" "users" {
  for_each = toset(["alice", "bob", "carol"])
  name     = each.key
}

# for_each với map of objects
resource "aws_route53_record" "records" {
  for_each = {
    api  = { ip = "10.0.1.1", ttl = 300 }
    www  = { ip = "10.0.1.2", ttl = 60  }
  }

  name    = each.key
```

```
  records = [each.value.ip]
  ttl     = each.value.ttl
}
```

Dùng `count` với list thì khi xóa phần tử giữa, Terraform sẽ recreate tất cả resource phía sau.

For expressions để transform data

```
locals {
  # List to map
  user_map = { for u in var.users : u.name => u.role }

  # Filter list
  prod_instances = [for i in var.instances : i if i.env == "prod"]

  # Transform map
  upper_tags = { for k, v in var.tags : k => upper(v) }
}
```

Flatten trick cho nested structures

```
variable "teams" {
  type = map(list(string))
  default = {
    backend  = ["alice", "bob"]
    frontend = ["carol", "dave"]
  }
}

locals {
  # Flatten thành list của objects
  team_members = flatten([
    for team, members in var.teams : [
      for member in members : {
        team   = team
        member = member
      }
    ]
  ])
}
# Kết quả: [{team="backend", member="alice"}, {team="backend", member="bob"}, ...]
```

Lifecycle Rules

create_before_destroy — zero-downtime replacement

```
resource "aws_instance" "web" {
  ami           = var.ami_id
  instance_type = var.instance_type

  lifecycle {
    create_before_destroy = true
  }
}
```

Terraform tạo resource mới trước, xong mới destroy cái cũ.

prevent_destroy — bảo vệ resource quan trọng

```
resource "aws_db_instance" "production" {
  lifecycle {
```

```
    prevent_destroy = true
  }
}
```

`prevent_destroy` chỉ ngăn `terraform destroy` và destroy trong plan — không ngăn được xóa thủ công trên console.

ignore_changes — bỏ qua thay đổi từ ngoài

```
resource "aws_autoscaling_group" "app" {
  desired_capacity = var.desired_capacity

  lifecycle {
    ignore_changes = [desired_capacity]  # auto-scaling tự điều chỉnh
  }
}
```

replace_triggered_by — force replacement khi dependency thay đổi

```
resource "aws_instance" "web" {
  lifecycle {
    replace_triggered_by = [
      aws_security_group.web.id,    # thay đổi SG → replace instance
    ]
  }
}
```

Drift Detection

Plan như là drift detector

```
# Chạy định kỳ để phát hiện drift
terraform plan -refresh-only -out=drift.tfplan
terraform show -json drift.tfplan | jq '.resource_changes[] | select(.change.actions != ["no-
op"])'
```

Reconcile external changes

```
# 1. Refresh state
terraform apply -refresh-only

# 2. Xem diff
terraform plan

# 3. Quyết định: import hoặc revert về Terraform config
```

Performance

Tăng parallelism

```
terraform apply -parallelism=20   # default là 10
```

Tăng quá cao có thể hit API rate limits của cloud provider.

I Provider caching

```
# ~/.terraformrc
provider_installation {
  filesystem_mirror {
    path    = "/usr/local/share/terraform/plugins"
    include = ["registry.terraform.io/*/*"]
  }
  direct {}
}
```

A Tối ưu large state

Khi state file lớn (>10MB), split thành nhiều state files theo layer.

```
states/
├── network/     # VPC, subnets, SG — thay đổi ít
├── data/        # RDS, ElastiCache — thay đổi ít
└── compute/     # EC2, ECS — thay đổi nhiều
```

Security

I Sensitive outputs

```
output "db_password" {
  value     = aws_db_instance.main.password
  sensitive = true
}
```

A OIDC auth thay vì long-lived credentials

```
# GitHub Actions với OIDC — không cần AWS_ACCESS_KEY
provider "aws" {
  region = "ap-southeast-1"
  # credentials tự động qua OIDC token
}
```

```
# GitHub Actions workflow
- name: Configure AWS credentials
  uses: aws-actions/configure-aws-credentials@v4
  with:
    role-to-assume: arn:aws:iam::123456789:role/github-actions-role
    aws-region: ap-southeast-1
```

A SOPS + Terraform cho secrets

```
data "sops_file" "secrets" {
  source_file = "secrets.enc.yaml"
}

resource "aws_db_instance" "main" {
  password = data.sops_file.secrets.data["db_password"]
}
```

! Secrets vẫn stored trong state file — encrypt state backend bắt buộc.

Testing

I Terraform test framework (TF 1.6+)

```
# tests/vpc.tftest.hcl
run "valid_vpc_creation" {
  command = plan

  assert {
    condition     = aws_vpc.main.cidr_block == "10.0.0.0/16"
    error_message = "VPC CIDR không đúng"
  }
}

run "apply_and_verify" {
  command = apply

  assert {
    condition     = output.vpc_id != ""
    error_message = "VPC ID không được rỗng"
  }
}
```

```
terraform test
terraform test -filter=tests/vpc.tftest.hcl
```

A Terratest — Go-based integration tests

```
func TestVpcModule(t *testing.T) {
    opts := &terraform.Options{
        TerraformDir: "../modules/vpc",
        Vars: map[string]interface{}{
            "environment": "test",
            "cidr_block":  "10.0.0.0/16",
        },
    }

    defer terraform.Destroy(t, opts)
    terraform.InitAndApply(t, opts)

    vpcId := terraform.Output(t, opts, "vpc_id")
    assert.NotEmpty(t, vpcId)
}
```

CI/CD Integration

I GitHub Actions cơ bản

```
name: Terraform CI

on:
  pull_request:
    branches: [main]

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: hashicorp/setup-terraform@v3
```

```
        with:
          terraform_version: "1.9.0"

      - name: Terraform Init
        run: terraform init

      - name: Terraform Format Check
        run: terraform fmt -check -recursive

      - name: Terraform Validate
        run: terraform validate

      - name: Terraform Plan
        run: terraform plan -out=tfplan.bin
        env:
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}

      - name: Post Plan to PR
        uses: borchero/terraform-plan-comment@v1
        with:
          plan-path: tfplan.bin
```

Atlantis PR workflow

```
# atlantis.yaml
version: 3
projects:
  - name: production
    dir: environments/prod
    workspace: default
    autoplan:
      when_modified: ["*.tf", "../modules/**/*.tf"]
    apply_requirements: [approved, mergeable]
```

Atlantis tự động `plan` khi có PR thay đổi `.tf` files, post kết quả vào PR comment, yêu cầu approval trước khi `apply`.

Atlantis cần network access vào cloud provider — deploy trong private network, không expose public.

Quick Reference

```
# Khởi tạo
terraform init -upgrade              # update providers
terraform init -backend-config=backend.hcl

# Plan/Apply
terraform plan -var-file=prod.tfvars -out=plan.bin
terraform apply plan.bin
terraform apply -auto-approve        # ! dùng cẩn thận

# Destroy
terraform destroy -target=module.old
terraform plan -destroy              # preview trước khi destroy

# State
terraform state list | grep module.vpc
terraform state show 'aws_instance.web[0]'
terraform state mv 'module.old' 'module.new'

# Debug
TF_LOG=DEBUG terraform plan 2>&1 | tee debug.log
terraform console
```

10

Ansible Tricks — Beginner to Advanced

> Giải thích bằng tiếng Việt. Commands/config bằng tiếng Anh. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

Inventory

B Static inventory cơ bản

```
# inventory/hosts.ini
[webservers]
web1 ansible_host=10.0.1.10
web2 ansible_host=10.0.1.11

[databases]
db1 ansible_host=10.0.2.10 ansible_user=ubuntu ansible_port=22

[production:children]
webservers
databases

[production:vars]
ansible_ssh_private_key_file=~/.ssh/prod_key
```

B host_vars và group_vars — biến theo host/group

```
inventory/
├── hosts.ini
├── group_vars/
│   ├── all.yml           # áp dụng cho tất cả hosts
│   ├── webservers.yml    # chỉ webservers
│   └── production/
│       ├── vars.yml      # biến non-sensitive
│       └── vault.yml     # biến sensitive (encrypted)
└── host_vars/
    ├── web1.yml
    └── db1.yml
```

```
# group_vars/webservers.yml
nginx_worker_processes: auto
nginx_worker_connections: 1024
app_port: 8080
```

I Dynamic inventory — lấy hosts từ cloud

```
# AWS EC2 dynamic inventory
ansible-inventory -i aws_ec2.yml --list
ansible-inventory -i aws_ec2.yml --graph
```

```yaml
# aws_ec2.yml
plugin: amazon.aws.aws_ec2
regions:
  - ap-southeast-1
filters:
  tag:Environment: production
  instance-state-name: running
keyed_groups:
  - key: tags.Role
    prefix: role
  - key: placement.region
    prefix: region
hostnames:
  - private-ip-address
```

I Constructed inventory — tạo group động từ facts

```yaml
# constructed.yml
plugin: ansible.builtin.constructed
groups:
  # Group theo OS
  ubuntu: ansible_distribution == "Ubuntu"
  centos: ansible_distribution == "CentOS"
  # Group theo environment tag
  web_prod: "'web' in group_names and 'production' in group_names"
compose:
  # Tạo variable mới từ existing vars
  ansible_user: "'ubuntu' if ansible_distribution == 'Ubuntu' else 'ec2-user'"
```

```
ansible-inventory -i aws_ec2.yml -i constructed.yml --graph
```

Playbook Patterns

B Handlers — chỉ restart khi có thay đổi

```yaml
# Handlers chỉ chạy khi được notify VÀ có changed task
tasks:
  - name: Copy nginx config
    ansible.builtin.template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify: Restart nginx

  - name: Copy SSL cert
    ansible.builtin.copy:
      src: cert.pem
      dest: /etc/ssl/cert.pem
    notify: Restart nginx   # cùng notify, handler chỉ chạy 1 lần

handlers:
  - name: Restart nginx
    ansible.builtin.service:
      name: nginx
      state: restarted
```

! Handler chạy ở cuối play, không phải ngay sau task notify. Dùng `meta: flush_handlers` để force chạy ngay.

B Tags — chạy subset của playbook

```yaml
tasks:
  - name: Install packages
```

```
    apt:
      name: nginx
    tags: [install, packages]

  - name: Configure nginx
    template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    tags: [configure, nginx]

  - name: Deploy app
    tags: [deploy]
    block:
      - copy:
          src: app.tar.gz
          dest: /opt/app/
      - command: /opt/app/deploy.sh
```

```
ansible-playbook site.yml --tags deploy
ansible-playbook site.yml --skip-tags install
ansible-playbook site.yml --list-tags    # xem tất cả tags
```

Blocks — group tasks với error handling

```
tasks:
  - name: Deploy application
    block:
      - name: Pull docker image
        community.docker.docker_image:
          name: myapp:latest
          source: pull

      - name: Run container
        community.docker.docker_container:
          name: myapp
          image: myapp:latest
          state: started

    rescue:
      - name: Notify on failure
        community.general.slack:
          token: "{{ slack_token }}"
          msg: "Deploy failed on {{ inventory_hostname }}"

      - name: Rollback to previous version
        community.docker.docker_container:
          name: myapp
          image: "myapp:{{ previous_version }}"
          state: started

    always:
      - name: Cleanup temp files
        file:
          path: /tmp/deploy
          state: absent
```

serial — rolling deploy theo batch

```
- hosts: webservers
  serial: 2          # deploy 2 hosts cùng lúc
  # serial: "30%"    # hoặc theo %
  # serial: [1, 3, 5]  # 1 host đầu, rồi 3, rồi 5 (canary pattern)

  tasks:
    - name: Deploy new version
      ...
```

I delegate_to — chạy task trên host khác

```yaml
tasks:
  - name: Remove from load balancer
    uri:
      url: "http://lb.internal/api/servers/{{ inventory_hostname }}/disable"
      method: POST
    delegate_to: localhost    # chạy trên control node

  - name: Deploy app
    ...

  - name: Add back to load balancer
    uri:
      url: "http://lb.internal/api/servers/{{ inventory_hostname }}/enable"
      method: POST
    delegate_to: localhost
```

I run_once — chạy đúng 1 lần cho cả group

```yaml
tasks:
  - name: Run database migration
    command: /app/manage.py migrate
    run_once: true          # chạy trên host đầu tiên trong group
    delegate_to: "{{ groups['app'][0] }}"  # hoặc chỉ định host cụ thể
```

Variables & Facts

B Thứ tự ưu tiên variable (thấp → cao)

```
1. role defaults
2. inventory file vars
3. inventory group_vars/all
4. inventory group_vars/*
5. inventory host_vars/*
6. playbook group_vars/all
7. playbook group_vars/*
8. playbook host_vars/*
9. host facts
10. play vars / vars_files
11. role vars
12. block vars
13. task vars
14. include_vars
15. set_fact / registered vars
16. extra vars (-e flag)  ← highest priority
```

B Register — lưu output của task

```yaml
tasks:
  - name: Check if service exists
    command: systemctl status myapp
    register: service_status
    failed_when: false    # không fail nếu service không tồn tại

  - name: Start service if not running
    service:
      name: myapp
      state: started
    when: service_status.rc != 0
```

I set_fact — tạo variable trong runtime

```yaml
tasks:
  - name: Get current date
    command: date +%Y%m%d
    register: date_output

  - name: Set deployment timestamp
    set_fact:
      deploy_timestamp: "{{ date_output.stdout }}"
      deploy_version: "{{ app_version }}-{{ date_output.stdout }}"
      cacheable: true    # cache fact cho các plays khác

  - name: Use the fact
    debug:
      msg: "Deploying version {{ deploy_version }}"
```

I Custom facts — facts từ script trên remote host

```
# Tạo file trên remote host: /etc/ansible/facts.d/app.fact
# File phải executable và output JSON

#!/bin/bash
echo '{"version": "1.2.3", "install_date": "2026-01-01"}'
```

```yaml
tasks:
  - name: Gather facts (bao gồm custom facts)
    setup:

  - name: Use custom fact
    debug:
      msg: "App version: {{ ansible_local.app.version }}"
```

B Magic variables quan trọng

```yaml
# inventory_hostname   — tên host trong inventory
# ansible_hostname     — hostname thực trên OS
# groups               — dict tất cả groups
# group_names          — list các group của host hiện tại
# hostvars             — facts của tất cả hosts
# playbook_dir         — thư mục chứa playbook
# role_path            — thư mục của role hiện tại

- name: Get IP of first db server
  debug:
    msg: "DB IP: {{ hostvars[groups['databases'][0]]['ansible_host'] }}"
```

Vault

B Encrypt/decrypt file

```
ansible-vault encrypt secrets.yml
ansible-vault decrypt secrets.yml
ansible-vault view secrets.yml        # xem không decrypt ra file
ansible-vault edit secrets.yml        # edit in-place
```

B Encrypt inline string

```
ansible-vault encrypt_string 'my_secret_password' --name db_password
# Output paste vào vars file:
```

```
# db_password: !vault |
#   $ANSIBLE_VAULT;1.1;AES256
#   ...
```

I Vault-id — nhiều password cho nhiều môi trường

```
# Tạo vault với id cụ thể
ansible-vault encrypt --vault-id prod@prompt secrets/prod.yml
ansible-vault encrypt --vault-id dev@~/.vault_pass_dev secrets/dev.yml

# Chạy playbook với nhiều vault password
ansible-playbook site.yml \
  --vault-id prod@prompt \
  --vault-id dev@~/.vault_pass_dev
```

I Rekey — đổi vault password

```
ansible-vault rekey secrets.yml
# hoặc
ansible-vault rekey --new-vault-password-file=new_pass.txt secrets.yml
```

A Multi-password vault — prod/dev riêng biệt

```
group_vars/
├── production/
│   ├── vars.yml      # non-sensitive
│   └── vault.yml     # encrypted với prod password
└── development/
    ├── vars.yml
    └── vault.yml     # encrypted với dev password
```

```
# vault password từ script (lấy từ secrets manager)
ansible-playbook site.yml --vault-id prod@get_vault_pass.sh
```

Roles

B Cấu trúc role chuẩn

```
roles/
└── nginx/
    ├── defaults/
    │   └── main.yml      # default variables (override được)
    ├── vars/
    │   └── main.yml      # internal variables (ít override hơn)
    ├── tasks/
    │   ├── main.yml      # entry point
    │   ├── install.yml
    │   └── configure.yml
    ├── handlers/
    │   └── main.yml
    ├── templates/
    │   └── nginx.conf.j2
    ├── files/
    │   └── ssl_params.conf
    ├── meta/
    │   └── main.yml      # role dependencies
    └── README.md
```

I defaults vs vars trong role

```
# defaults/main.yml — có thể override từ inventory/playbook
nginx_port: 80
nginx_worker_processes: auto

# vars/main.yml — ưu tiên cao hơn, dùng cho internal logic
nginx_pid_file: /var/run/nginx.pid
nginx_config_dir: /etc/nginx
```

I include_role vs import_role

```
# import_role: static, processed tại parse time
# - tags áp dụng cho tất cả tasks trong role
# - không thể dùng với loop
- import_role:
    name: common

# include_role: dynamic, processed tại runtime
# - có thể dùng với loop và when
# - tags chỉ áp dụng cho include task
- include_role:
    name: "{{ item }}"
  loop:
    - nginx
    - php-fpm
```

I Role dependencies

```
# meta/main.yml
dependencies:
  - role: common
    vars:
      ntp_server: time.cloudflare.com
  - role: security-hardening
    when: ansible_os_family == "Debian"
```

B Cài role từ Galaxy/Collections

```
ansible-galaxy install geerlingguy.nginx
ansible-galaxy collection install community.docker
ansible-galaxy install -r requirements.yml

# requirements.yml
# roles:
#   - name: geerlingguy.nginx
#     version: "3.2.0"
# collections:
#   - name: community.docker
#     version: "≥3.0.0"
```

Modules & Plugins

B Modules hay dùng nhất

```
# File operations
ansible.builtin.file:          # create/delete files, dirs, symlinks
ansible.builtin.copy:          # copy local file lên remote
ansible.builtin.template:      # render Jinja2 template lên remote
ansible.builtin.fetch:         # download file từ remote về local
ansible.builtin.lineinfile:    # thêm/sửa/xóa dòng trong file
```

```
# Package management
ansible.builtin.apt:           # Ubuntu/Debian
ansible.builtin.yum:           # CentOS/RHEL
ansible.builtin.package:       # generic, tự detect package manager

# Service
ansible.builtin.service:       # start/stop/restart/enable services
ansible.builtin.systemd:       # systemd-specific (daemon_reload, etc.)

# Commands
ansible.builtin.command:       # chạy command, không qua shell
ansible.builtin.shell:         # chạy qua shell (có pipe, redirect)
ansible.builtin.script:        # copy + chạy script local trên remote

# Users/Groups
ansible.builtin.user:
ansible.builtin.group:
ansible.builtin.authorized_key:

# Network
ansible.builtin.uri:           # HTTP requests
ansible.builtin.get_url:       # download file từ URL
ansible.builtin.wait_for:      # đợi port/file/condition
```

I Lookup plugins — đọc data từ control node

```
vars:
  # Đọc file local
  ssl_cert: "{{ lookup('file', 'files/cert.pem') }}"

  # Đọc environment variable
  db_password: "{{ lookup('env', 'DB_PASSWORD') }}"

  # Đọc từ password file (tạo nếu chưa có)
  app_secret: "{{ lookup('password', 'credentials/app_secret length=32') }}"

  # Query DNS
  server_ip: "{{ lookup('dig', 'example.com') }}"
```

I Callback plugins — customize output

```
# ansible.cfg
[defaults]
stdout_callback = yaml          # output dạng YAML, dễ đọc hơn
# stdout_callback = json        # machine-readable
# stdout_callback = minimal     # tối giản
callbacks_enabled = timer, profile_tasks   # (callback_whitelist đã removed từ ansible-core 2.15)
```

Templates

B Jinja2 cơ bản trong template

```
{# nginx.conf.j2 #}
worker_processes {{ nginx_worker_processes | default('auto') }};

http {
    server {
        listen {{ nginx_port }};
        server_name {{ ansible_fqdn }};

        {# Conditional block #}
        {% if ssl_enabled %}
```

```
        listen 443 ssl;
        ssl_certificate {{ ssl_cert_path }};
        {% endif %}

        {# Loop #}
        {% for location in nginx_locations %}
        location {{ location.path }} {
            proxy_pass {{ location.upstream }};
        }
        {% endfor %}
    }
}
```

I Jinja2 filters hữu ích

```
{# String filters #}
{{ hostname | upper }}
{{ path | basename }}
{{ path | dirname }}
{{ "hello world" | replace(" ", "_") }}

{# List/dict filters #}
{{ users | join(', ') }}
{{ items | sort }}
{{ items | unique }}
{{ dict | dict2items }}        {# dict → list of {key, value} #}
{{ list | items2dict }}        {# reverse #}
{{ items | selectattr('active', 'truthy') | list }}
{{ items | map(attribute='name') | list }}

{# Default và defined #}
{{ var | default('fallback') }}
{{ var | default(omit) }}      {# bỏ qua param nếu undefined #}

{# Type conversion #}
{{ "42" | int }}
{{ value | string }}
{{ value | bool }}
{{ value | to_json }}
{{ value | to_yaml }}
{{ value | to_nice_json(indent=2) }}
```

I Template validation trước khi deploy

```
- name: Deploy nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
    validate: nginx -t -c %s    # %s = path đến temp file
  notify: Reload nginx
```

Loops & Conditionals

B Loop cơ bản

```
- name: Install packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - python3
    - git
```

```
- name: Create users
  user:
    name: "{{ item.name }}"
    groups: "{{ item.groups }}"
    state: present
  loop:
    - { name: alice, groups: "sudo,docker" }
    - { name: bob,   groups: "docker" }
```

until — retry cho đến khi thành công

```
- name: Wait for app to be ready
  uri:
    url: http://localhost:8080/health
    status_code: 200
  register: health_check
  until: health_check.status == 200
  retries: 10
  delay: 5          # giây giữa mỗi lần retry
```

when — conditionals

```
- name: Install on Debian only
  apt:
    name: nginx
  when: ansible_os_family == "Debian"

- name: Deploy to prod
  include_tasks: deploy_prod.yml
  when:
    - env == "production"
    - not maintenance_mode | bool

- name: Skip if already done
  command: /opt/app/migrate.sh
  when: migration_needed is defined and migration_needed
```

selectattr trong loop — filter list of dicts

```
vars:
  users:
    - { name: alice, active: true,  role: admin }
    - { name: bob,   active: false, role: user  }
    - { name: carol, active: true,  role: user  }

tasks:
  - name: Create only active users
    user:
      name: "{{ item.name }}"
    loop: "{{ users | selectattr('active') | list }}"

  - name: Get admin names
    debug:
      msg: "{{ users | selectattr('role', 'eq', 'admin') | map(attribute='name') | list }}"
```

Register trong loop

```
- name: Check service status
  command: systemctl is-active {{ item }}
  register: service_results
  failed_when: false
  loop:
    - nginx
    - mysql
```

```
    - redis

- name: Show failed services
  debug:
    msg: "Service {{ item.item }} is NOT running"
  loop: "{{ service_results.results }}"
  when: item.rc != 0
```

Performance

I SSH pipelining — giảm số SSH connections

```
# ansible.cfg
[ssh_connection]
pipelining = true
```

! Cần tắt `requiretty` trong sudoers trên remote hosts.

I SSH multiplexing — tái dùng connection

```
# ansible.cfg
[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
control_path = /tmp/ansible-ssh-%%h-%%p-%%r
```

I Fact caching — cache facts giữa các lần chạy

```
# ansible.cfg
[defaults]
gathering = smart           # chỉ gather nếu chưa có trong cache
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible-facts-cache
fact_caching_timeout = 3600
```

A Strategy: free — hosts không chờ nhau

```
- hosts: webservers
  strategy: free     # mỗi host chạy độc lập, không đồng bộ

  tasks:
    - ...
```

Mặc định là `linear` — tất cả hosts xong task N thì mới sang task N+1.

A Async — chạy task nền

```
- name: Long running operation
  command: /opt/scripts/long-task.sh
  async: 300        # timeout 300 giây
  poll: 0           # không đợi, trả về ngay

- name: Check task status
  async_status:
    jid: "{{ long_task.ansible_job_id }}"
  register: task_result
  until: task_result.finished
  retries: 30
  delay: 10
```

A Mitogen — accelerate Ansible đáng kể

```
# ansible.cfg
[defaults]
strategy_plugins = /path/to/mitogen/ansible_mitogen/plugins/strategy
strategy = mitogen_linear
```

Mitogen thường nhanh hơn default 2-7x bằng cách tối ưu Python execution channel.

Error Handling

B ignore_errors và failed_when

```
- name: Check if process exists
  command: pgrep myapp
  register: pgrep_result
  ignore_errors: true    # không dừng play nếu fail

- name: Custom fail condition
  command: /opt/healthcheck.sh
  register: hc
  failed_when:
    - hc.rc != 0
    - "'CRITICAL' in hc.stdout"    # chỉ fail nếu có chữ CRITICAL
```

I changed_when — control khi nào task là "changed"

```
- name: Run idempotent script
  command: /opt/app/setup.sh
  register: setup_result
  changed_when: "'already configured' not in setup_result.stdout"

- name: Always show as OK (never changed)
  command: /opt/app/status.sh
  changed_when: false
```

I any_errors_fatal và max_fail_percentage

```
- hosts: webservers
  any_errors_fatal: true    # dừng tất cả nếu bất kỳ host nào fail

# Hoặc cho phép một số % host fail
- hosts: webservers
  max_fail_percentage: 20   # dừng nếu > 20% hosts fail
```

Debugging

B Verbose modes

```
ansible-playbook site.yml -v      # verbose
ansible-playbook site.yml -vv     # more verbose (files, connections)
ansible-playbook site.yml -vvv    # SSH debug
ansible-playbook site.yml -vvvv   # connection plugin debug
```

B Debug module

```
- name: Print variable
  debug:
    msg: "Value is: {{ my_var }}"
    verbosity: 2    # chỉ hiện khi chạy với -vv

- name: Dump all variables
  debug:
    var: hostvars[inventory_hostname]

- name: Print multiple vars
  debug:
    msg:
      - "OS: {{ ansible_distribution }}"
      - "IP: {{ ansible_default_ipv4.address }}"
```

I Assert — fail với message rõ ràng

```
- name: Verify preconditions
  assert:
    that:
      - ansible_memtotal_mb ⩾ 2048
      - ansible_distribution in ['Ubuntu', 'Debian']
      - app_version is defined
    fail_msg: "Precondition failed — check requirements"
    success_msg: "All preconditions met"
```

I ansible-lint — lint playbooks

```
ansible-lint site.yml
ansible-lint roles/nginx/
ansible-lint --list-rules
```

A Molecule — test roles trong container

```
# Khởi tạo molecule trong role
molecule init scenario

# Test workflow
molecule create      # tạo container/VM
molecule converge    # chạy role
molecule verify      # chạy tests
molecule destroy     # xóa container

# All in one
molecule test
```

```
# molecule/default/verify.yml
- name: Verify nginx
  hosts: all
  tasks:
    - name: Check nginx running
      service_facts:

    - name: Assert nginx is active
      assert:
        that: ansible_facts.services['nginx'].state == 'running'
```

Idempotency

B command vs shell — dùng creates/removes

```
# ❗ Không idempotent — chạy mỗi lần
- command: tar xzf app.tar.gz -C /opt/app

# Idempotent — skip nếu file đã tồn tại
- command: tar xzf app.tar.gz -C /opt/app
  args:
    creates: /opt/app/bin/server    # skip nếu file này tồn tại
    removes: /tmp/app.tar.gz        # chỉ chạy nếu file này tồn tại
```

B stat + when pattern — check trước khi làm

```
- name: Check if app is installed
  stat:
    path: /opt/app/bin/server
  register: app_binary

- name: Install app (only if not present)
  unarchive:
    src: app.tar.gz
    dest: /opt/app/
  when: not app_binary.stat.exists

- name: Get current version
  command: /opt/app/bin/server --version
  register: current_version
  when: app_binary.stat.exists
  changed_when: false
```

I Dùng module thay vì shell

```
# ❗ Không idempotent
- shell: useradd -m -s /bin/bash alice

# Idempotent
- user:
    name: alice
    shell: /bin/bash
    create_home: true
    state: present

# ❗ Không idempotent
- shell: echo "net.ipv4.ip_forward=1" >> /etc/sysctl.conf

# Idempotent
- lineinfile:
    path: /etc/sysctl.conf
    line: "net.ipv4.ip_forward=1"
    regexp: "^net.ipv4.ip_forward"
```

Advanced Patterns

A Rolling update với pre/post tasks

```
- name: Rolling deployment
  hosts: webservers
  serial: 1         # deploy từng host một
```

```
  pre_tasks:
    - name: Remove from load balancer
      uri:
        url: "http://lb.internal/drain/{{ inventory_hostname }}"
        method: POST
      delegate_to: localhost

    - name: Wait for connections to drain
      wait_for:
        timeout: 30

  tasks:
    - name: Deploy new version
      include_role:
        name: app-deploy

  post_tasks:
    - name: Health check
      uri:
        url: "http://{{ ansible_host }}:8080/health"
        status_code: 200
      retries: 5
      delay: 10

    - name: Add back to load balancer
      uri:
        url: "http://lb.internal/enable/{{ inventory_hostname }}"
        method: POST
      delegate_to: localhost
```

A Dynamic includes — chọn tasks theo runtime condition

```
- name: Include OS-specific tasks
  include_tasks: "{{ ansible_distribution | lower }}.yml"
  # Sẽ include: ubuntu.yml hoặc centos.yml tùy OS

- name: Include environment-specific config
  include_vars: "{{ item }}"
  with_first_found:
    - "{{ env }}_{{ region }}.yml"    # prod_ap-southeast-1.yml
    - "{{ env }}.yml"                  # prod.yml
    - "default.yml"                    # fallback
```

A Inventory refresh mid-play — thêm host dynamic

```
- hosts: localhost
  tasks:
    - name: Provision new server
      ec2_instance:
        name: new-web-server
        ...
      register: new_instance

    - name: Add new server to inventory
      add_host:
        name: "{{ new_instance.instances[0].public_ip }}"
        groups: newly_provisioned
        ansible_user: ubuntu

- hosts: newly_provisioned
  tasks:
    - name: Configure new server
      ...
```

Quick Reference

```
# Ad-hoc commands
ansible all -m ping
ansible webservers -m command -a "uptime"
ansible webservers -m shell -a "df -h | grep /dev/sda"
ansible web1 -m setup | grep ansible_distribution
ansible all -m apt -a "name=nginx state=present" --become

# Playbook options
ansible-playbook site.yml --check           # dry-run
ansible-playbook site.yml --diff            # show file diffs
ansible-playbook site.yml --check --diff    # cả hai
ansible-playbook site.yml --start-at-task="Deploy app"
ansible-playbook site.yml --step            # confirm từng task
ansible-playbook site.yml -e "env=prod version=1.2.3"
ansible-playbook site.yml -l webservers     # limit hosts

# Inventory
ansible-inventory --list
ansible-inventory --graph
ansible-inventory --host web1

# Vault
ansible-vault encrypt_string 'secret' --name myvar
ansible-playbook site.yml --ask-vault-pass
ansible-playbook site.yml --vault-password-file ~/.vault_pass

# Galaxy
ansible-galaxy role install geerlingguy.nginx
ansible-galaxy collection install community.docker
ansible-galaxy role list
```

11

Prometheus & Grafana — DevOps Tricks

> Practical reference từ beginner đến advanced. Commands/config bằng tiếng Anh, giải thích bằng tiếng Việt. Tags: B beginner · I intermediate · A advanced · ! gotcha

---

PROMETHEUS

PromQL Basics

B **Instant vector selector** Lấy giá trị hiện tại của một metric, có thể lọc theo label.

```
# Lấy tất cả time series của metric này
http_requests_total

# Lọc theo label chính xác (equality matcher)
http_requests_total{job="api", status="200"}

# Lọc theo regex (=~)
http_requests_total{status=~"5.."}

# Loại trừ theo regex (≠, !~)
http_requests_total{status!~"2.."}
```

B **Range vector selector** Lấy tất cả data points trong một khoảng thời gian — bắt buộc dùng với các hàm như `rate()`, `increase()`.

```
# Lấy data trong 5 phút gần nhất
http_requests_total[5m]

# Các đơn vị thời gian: ms, s, m, h, d, w, y
node_cpu_seconds_total[1h]
```

B **Offset modifier** So sánh metric hiện tại với quá khứ.

```
# Request rate hiện tại so với 1 tuần trước
rate(http_requests_total[5m]) / rate(http_requests_total[5m] offset 1w)
```

! Offset phải đặt SAU selector, không phải sau function.

```
# Sai
rate(offset 1w http_requests_total[5m])

# Đúng
rate(http_requests_total[5m] offset 1w)
```

PromQL Advanced

rate() vs irate() `rate()` tính tốc độ trung bình qua toàn bộ range. `irate()` chỉ dùng 2 data point cuối — nhạy cảm hơn với spike.

```
# Dùng rate() cho dashboard dài hạn, alerting
rate(http_requests_total[5m])

# Dùng irate() khi cần phát hiện spike tức thì
irate(http_requests_total[5m])
```

`irate()` bị ảnh hưởng nhiều bởi scrape jitter. Không dùng cho alert rules vì dễ false positive.

histogram_quantile Tính phần vị từ histogram metric (p50, p95, p99 latency).

```
# p99 latency của HTTP requests
histogram_quantile(0.99, rate(http_request_duration_seconds_bucket[5m]))

# p99 latency group by service
histogram_quantile(0.99,
  sum by (le, service) (
    rate(http_request_duration_seconds_bucket[5m])
  )
)
```

Phải `sum by (le, ...)` — nếu bỏ `le` thì histogram_quantile trả về NaN.

predict_linear Dự đoán giá trị trong tương lai dựa trên xu hướng hiện tại. Dùng để cảnh báo disk full sớm.

```
# Dự đoán disk sẽ đầy trong bao nhiêu giây nữa (nếu âm = đã đầy)
predict_linear(node_filesystem_avail_bytes[1h], 4 * 3600) < 0

# Alert: disk sẽ đầy trong 24h
predict_linear(node_filesystem_avail_bytes[6h], 24 * 3600) < 0
```

Aggregation operators Gộp nhiều time series lại theo label.

```
# Tổng request theo status code
sum by (status) (rate(http_requests_total[5m]))

# Loại bỏ job label, giữ lại các label khác
sum without (job) (http_requests_total)

# Top 5 service có nhiều request nhất
topk(5, sum by (service) (rate(http_requests_total[5m])))

# Số lượng instance đang up
count by (job) (up == 1)
```

Subqueries Chạy range query bên trong instant query — dùng khi muốn tính max/min của rate() qua một khoảng thời gian.

```
# Max request rate trong 1 giờ qua, bước 1 phút
max_over_time(rate(http_requests_total[5m])[1h:1m])

# Alert nếu p99 latency vượt 500ms trong 30 phút qua
max_over_time(
  histogram_quantile(0.99, rate(http_request_duration_seconds_bucket[5m]))[30m:1m]
) > 0.5
```

Subquery nặng — tránh dùng trong dashboard có nhiều panel. Dùng recording rules thay thế.

Recording Rules

Tại sao dùng recording rules Pre-compute các query nặng, kết quả được lưu thành metric mới. Dashboard load nhanh hơn, alert evaluate nhanh hơn.

Naming convention Format chuẩn: `level:metric:operations`

```yaml
# prometheus/rules/recording.yml
groups:
  - name: http_recordings
    interval: 1m  # Tần suất tính toán, mặc định = global evaluation_interval
    rules:
      # level = job, metric = http_requests_total, operation = rate5m
      - record: job:http_requests_total:rate5m
        expr: sum by (job) (rate(http_requests_total[5m]))

      # p99 latency pre-computed
      - record: job:http_request_duration_seconds:p99_rate5m
        expr: |
          histogram_quantile(0.99,
            sum by (job, le) (rate(http_request_duration_seconds_bucket[5m]))
          )
```

Rule groups — performance gain Mỗi group có thể chạy song song. Group cùng loại metric nên nhóm lại để tránh dependency issue.

```yaml
groups:
  - name: node_recordings     # Chạy song song với group khác
    interval: 30s
    rules:
      - record: instance:node_cpu_utilisation:rate5m
        expr: |
          1 - avg by (instance) (
            rate(node_cpu_seconds_total{mode="idle"}[5m])
          )

      - record: instance:node_memory_utilisation:ratio
        expr: |
          1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)
```

Recording rule không được tham chiếu chính nó (circular dependency). Kiểm tra bằng `promtool check rules`.

Alerting

Alert rule cơ bản

```yaml
# prometheus/rules/alerts.yml
groups:
  - name: service_alerts
    rules:
      - alert: HighErrorRate
        expr: |
          (
            sum by (job) (rate(http_requests_total{status=~"5.."}[5m]))
            /
            sum by (job) (rate(http_requests_total[5m]))
          ) > 0.05
        for: 5m   # Phải đúng liên tục 5 phút mới fire
        labels:
          severity: critical
          team: backend
        annotations:
          summary: "High error rate on {{ $labels.job }}"
          description: "Error rate is {{ $value | humanizePercentage }} for {{ $labels.job }}"
```

`for: 5m` nghĩa là alert ở trạng thái `pending` 5 phút trước khi chuyển sang `firing`. Không có `for` thì alert fire ngay lập tức — dễ flap.

Alertmanager routing tree

```yaml
# alertmanager/alertmanager.yml
global:
  resolve_timeout: 5m
  slack_api_url: 'https://hooks.slack.com/...'

route:
  receiver: 'default'
  group_by: ['alertname', 'job']
  group_wait: 30s      # Đợi trước khi gửi notification group đầu tiên
  group_interval: 5m   # Gửi notification update sau mỗi 5 phút
  repeat_interval: 4h  # Re-notify nếu alert vẫn firing sau 4h

  routes:
    - matchers:
        - severity = critical
      receiver: pagerduty
      continue: false  # Không tiếp tục match route khác

    - matchers:
        - team = backend
      receiver: slack-backend

receivers:
  - name: pagerduty
    pagerduty_configs:
      - routing_key: '<key>'
  - name: slack-backend
    slack_configs:
      - channel: '#backend-alerts'
        title: '{{ .GroupLabels.alertname }}'
```

Inhibition — tắt alert con khi alert cha đang firing

```yaml
inhibit_rules:
  # Nếu node down thì tắt hết alert khác trên node đó
  - source_matchers:
      - alertname = NodeDown
    target_matchers:
      - severity =~ "warning|info"
    equal: ['instance']  # Phải match cùng instance label
```

Silence via API

```bash
# Tạo silence trong 2 giờ qua API
curl -X POST http://alertmanager:9093/api/v2/silences \
  -H 'Content-Type: application/json' \
  -d '{
    "matchers": [{"name": "job", "value": "api", "isRegex": false}],
    "startsAt": "2024-01-01T00:00:00Z",
    "endsAt": "2024-01-01T02:00:00Z",
    "comment": "Deployment window",
    "createdBy": "ops-team"
  }'
```

Service Discovery

Static config

```yaml
scrape_configs:
  - job_name: 'api'
```

```
    static_configs:
      - targets: ['api-1:8080', 'api-2:8080']
        labels:
          env: production
```

File-based SD — dynamic targets không cần restart

```
scrape_configs:
  - job_name: 'dynamic-targets'
    file_sd_configs:
      - files:
          - '/etc/prometheus/targets/*.json'
        refresh_interval: 30s  # Reload file mỗi 30s
```

```
# /etc/prometheus/targets/api.json
[{
  "targets": ["api-1:8080", "api-2:8080"],
  "labels": {"env": "prod", "team": "backend"}
}]
```

Kubernetes SD

```
scrape_configs:
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      # Chỉ scrape pod có annotation prometheus.io/scrape: "true"
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
        action: keep
        regex: 'true'
      # Lấy port từ annotation
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_port]
        action: replace
        target_label: __address__
        regex: (.+)
        replacement: '${__meta_kubernetes_pod_ip}:$1'
```

Relabeling

relabel_configs vs metric_relabel_configs - `relabel_configs`: chạy TRƯỚC khi scrape, dùng để quyết định có scrape không, sửa target address - `metric_relabel_configs`: chạy SAU khi scrape, dùng để lọc/sửa metric

```
scrape_configs:
  - job_name: 'node'
    relabel_configs:
      # Đổi __address__ thành dùng port 9100
      - source_labels: [__address__]
        regex: '([^:]+).*'
        replacement: '$1:9100'
        target_label: __address__

      # Drop target có label environment=dev
      - source_labels: [__meta_consul_service_metadata_environment]
        regex: dev
        action: drop

    metric_relabel_configs:
      # Drop metric có tên chứa "debug"
      - source_labels: [__name__]
        regex: '.*debug.*'
        action: drop

      # Đổi tên label
      - source_labels: [exported_job]
```

```
        target_label: job
        action: replace
```

A Hashmod — sharding Prometheus

```
# Prometheus shard 0/3: chỉ scrape 1/3 targets
relabel_configs:
  - source_labels: [__address__]
    modulus: 3
    target_label: __tmp_hash
    action: hashmod
  - source_labels: [__tmp_hash]
    regex: '^0$'
    action: keep
```

Storage

B Retention config

```
# prometheus.yml hoặc CLI flag
# Giữ data 30 ngày hoặc tối đa 50GB, cái nào đến trước
--storage.tsdb.retention.time=30d
--storage.tsdb.retention.size=50GB
```

I Remote write — gửi data sang long-term storage

```
remote_write:
  - url: 'https://thanos-receive:19291/api/v1/receive'
    queue_config:
      max_samples_per_send: 10000
      max_shards: 30
      capacity: 100000
    write_relabel_configs:
      # Chỉ gửi metric quan trọng sang remote
      - source_labels: [__name__]
        regex: 'job:.*|instance:.*'
        action: keep
```

A Thanos basics — HA + long-term storage

```
# thanos-sidecar chạy cạnh Prometheus
# Upload TSDB blocks lên object storage (S3, GCS)
thanos sidecar \
  --tsdb.path=/prometheus \
  --objstore.config-file=/etc/thanos/s3.yml \
  --prometheus.url=http://localhost:9090

# thanos-query — query từ nhiều Prometheus
thanos query \
  --store=thanos-sidecar-1:10901 \
  --store=thanos-sidecar-2:10901 \
  --store=thanos-store:10901  # Historical data từ S3
```

Exporters

B node_exporter — host metrics

```
# Chạy node_exporter
docker run -d \
  --name node-exporter \
  --pid="host" \
  --network="host" \
  -v /:/host:ro,rslave \
```

```
  prom/node-exporter \
  --path.rootfs=/host \
  --collector.filesystem.mount-points-exclude='^/(sys|proc|dev|host|etc)($$|/)'
```

I blackbox_exporter — probe external endpoints

```yaml
# blackbox.yml
modules:
  http_2xx:
    prober: http
    timeout: 5s
    http:
      valid_status_codes: [200, 201, 204]
      method: GET
      tls_config:
        insecure_skip_verify: false

# prometheus.yml — scrape blackbox để probe các URL
scrape_configs:
  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
          - https://api.example.com/health
          - https://app.example.com
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: blackbox-exporter:9115
```

I Custom metrics với Pushgateway Dùng cho batch jobs — không nên dùng cho long-running services.

```
# Push metric từ shell script
echo "batch_job_duration_seconds 42.3" | \
  curl --data-binary @- \
  http://pushgateway:9091/metrics/job/backup/instance/server-1

# Push nhiều metric
cat <<EOF | curl --data-binary @- http://pushgateway:9091/metrics/job/backup
# TYPE backup_files_total counter
backup_files_total{type="incremental"} 1523
# TYPE backup_size_bytes gauge
backup_size_bytes 9.3e+09
EOF
```

! Pushgateway không tự xóa metric khi job kết thúc. Phải gọi DELETE sau khi job xong, nếu không metric cũ sẽ tồn tại mãi.

Performance

A Giảm cardinality — vấn đề phổ biến nhất

```
# Xem top 10 metric theo số lượng series
curl -s http://localhost:9090/api/v1/status/tsdb | \
  jq '.data.seriesCountByMetricName[:10]'

# Xem label có cardinality cao nhất
curl -s http://localhost:9090/api/v1/status/tsdb | \
  jq '.data.labelValueCountByLabelName[:10]'
```

```
# Xóa label có quá nhiều giá trị (ví dụ: user_id, request_id)
metric_relabel_configs:
  - regex: 'user_id|request_id|trace_id'
    action: labeldrop
```

I Scrape interval tuning

```
global:
  scrape_interval: 15s      # Mặc định, phù hợp cho hầu hết
  evaluation_interval: 15s  # Tần suất evaluate rules

scrape_configs:
  - job_name: 'fast-metrics'   # Metric thay đổi nhanh
    scrape_interval: 5s
  - job_name: 'slow-metrics'   # Metric ổn định, tiết kiệm storage
    scrape_interval: 60s
```

GRAFANA

Dashboard Design

B Variables/Templates — dashboard động

```
// Định nghĩa variable trong dashboard JSON
{
  "templating": {
    "list": [{
      "name": "datasource",
      "type": "datasource",
      "query": "prometheus"
    }, {
      "name": "job",
      "type": "query",
      "datasource": "$datasource",
      "query": "label_values(up, job)",
      "refresh": 2,          // Refresh khi time range thay đổi
      "multi": true,         // Cho phép chọn nhiều
      "includeAll": true,    // Option "All"
      "allValue": ".+"       // Regex match tất cả
    }]
  }
}
```

```
# Dùng variable trong query
rate(http_requests_total{job=~"$job"}[5m])
```

I Repeat panels theo variable Tự động tạo panel cho mỗi giá trị của variable — dùng để hiển thị từng service/instance riêng.

Trong panel settings: `Repeat by variable` → chọn variable `job`. Direction: Horizontal (tối đa 6 panels/row) hoặc Vertical.

I Annotations — đánh dấu event trên graph

```
{
  "annotations": {
    "list": [{
      "datasource": "-- Grafana --",
      "enable": true,
```

```
      "name": "Deployments",
      "type": "dashboard"
    }, {
      "datasource": "Prometheus",
      "enable": true,
      "expr": "changes(deploy_timestamp[1m]) > 0",
      "name": "Auto Deploy Annotations",
      "titleFormat": "Deploy: {{ $labels.version }}"
    }]
  }
}
```

Query Patterns

I Transformations — xử lý data trong Grafana

```
# Merge nhiều query thành 1 table
Query A: sum by (job) (rate(http_requests_total[5m]))
Query B: sum by (job) (rate(http_errors_total[5m]))

Transformations:
1. Merge (gộp A và B theo field chung là job)
2. Organize fields (đổi tên cột, sắp xếp)
3. Add field from calculation (B/A = error rate)
```

I Table với thresholds

```
{
  "fieldConfig": {
    "overrides": [{
      "matcher": {"id": "byName", "options": "Error Rate"},
      "properties": [{
        "id": "thresholds",
        "value": {
          "mode": "absolute",
          "steps": [
            {"color": "green", "value": null},
            {"color": "yellow", "value": 0.01},
            {"color": "red", "value": 0.05}
          ]
        }
      }, {
        "id": "custom.displayMode",
        "value": "color-background"
      }]
    }]
  }
}
```

A Mixing data sources trong 1 panel

```
# Prometheus: latency metric
histogram_quantile(0.99, rate(http_request_duration_seconds_bucket[5m]))

# Loki: query log count (data source riêng)
sum(count_over_time({job="api"} |= "ERROR" [5m]))

# Kết hợp bằng Transformations > Join by field (time)
```

Alerting

I Unified Alerting (Grafana 9+)

```yaml
# Grafana alert rule (provisioning)
apiVersion: 1
groups:
  - orgId: 1
    name: HTTP Alerts
    folder: Production
    interval: 1m
    rules:
      - uid: high-error-rate
        title: High Error Rate
        condition: C
        data:
          - refId: A
            datasourceUid: prometheus
            model:
              expr: sum(rate(http_requests_total{status=~"5.."}[5m]))
          - refId: B
            datasourceUid: prometheus
            model:
              expr: sum(rate(http_requests_total[5m]))
          - refId: C        # Condition
            datasourceUid: __expr__
            model:
              type: math
              expression: "$A / $B > 0.05"
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: Error rate exceeded 5%
```

Contact points & notification policies

```yaml
# Provisioning: contact points
apiVersion: 1
contactPoints:
  - orgId: 1
    name: Slack-Ops
    receivers:
      - uid: slack-ops-uid
        type: slack
        settings:
          url: https://hooks.slack.com/...
          channel: '#ops-alerts'
          title: '{{ template "slack.title" . }}'

# Notification policy
policies:
  - orgId: 1
    receiver: Slack-Ops
    group_by: ['grafana_folder', 'alertname']
    routes:
      - receiver: PagerDuty
        matchers:
          - severity = critical
        group_wait: 0s
```

Provisioning

Dashboard as code

```yaml
# /etc/grafana/provisioning/dashboards/default.yml
apiVersion: 1
providers:
  - name: default
    type: file
    updateIntervalSeconds: 30
```

```
    options:
      path: /var/lib/grafana/dashboards
      foldersFromFilesStructure: true  # Tạo folder từ thư mục con
```

```
# Export dashboard hiện tại thành JSON
curl -s http://admin:admin@localhost:3000/api/dashboards/uid/<uid> | \
  jq '.dashboard' > dashboard.json
```

I Datasource provisioning

```
# /etc/grafana/provisioning/datasources/prometheus.yml
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
    isDefault: true
    jsonData:
      timeInterval: 15s
      exemplarTraceIdDestinations:
        - name: traceID
          datasourceUid: tempo
```

A Grafonnet — dashboard as code với Jsonnet

```
// dashboard.jsonnet
local grafana = import 'grafonnet/grafana.libsonnet';
local dashboard = grafana.dashboard;
local graphPanel = grafana.graphPanel;
local prometheus = grafana.prometheus;

dashboard.new(
  'HTTP Overview',
  tags=['http', 'production'],
  time_from='now-1h',
)
.addPanel(
  graphPanel.new('Request Rate')
  .addTarget(
    prometheus.target(
      'rate(http_requests_total[5m])',
      legendFormat='{{job}}',
    )
  ),
  gridPos={x: 0, y: 0, w: 12, h: 8}
)
```

```
# Compile Jsonnet thành JSON
jsonnet -J vendor dashboard.jsonnet > dashboard.json
```

Advanced

I Loki integration — logs trong Grafana

```
# Query Loki trong Grafana panel
# Hiển thị log errors
{job="api", env="production"} |= "ERROR" | json | line_format "{{.message}}"

# Count errors theo service
sum by (service) (
  count_over_time({job="api"} |= "ERROR" [5m])
)
```

```
# Correlate metrics + logs: dùng Explore > split view
# Hoặc dùng "Logs" panel type với Loki datasource
```

Public dashboards & snapshots

```
# Tạo snapshot (không cần auth để xem)
curl -X POST http://admin:admin@localhost:3000/api/snapshots \
  -H 'Content-Type: application/json' \
  -d '{
    "dashboard": <dashboard_json>,
    "expires": 3600,
    "name": "Incident 2024-01-15"
  }'
```

Embedding dashboard trong app khác

```
<!-- Embed panel vào iframe -->
<iframe
  src="https://grafana.example.com/d-solo/<uid>/<slug>?
orgId=1&panelId=2&from=now-1h&to=now&theme=light"
  width="800"
  height="400"
  frameborder="0"
></iframe>
```

Cần enable `allow_embedding = true` trong `grafana.ini` và cấu hình `cookie_samesite = none`.

```
[security]
allow_embedding = true
cookie_samesite = none
```

Playlist — auto-rotate dashboards cho TV/NOC

```
# Tạo playlist qua API
curl -X POST http://admin:admin@localhost:3000/api/playlists \
  -H 'Content-Type: application/json' \
  -d '{
    "name": "NOC Overview",
    "interval": "5m",
    "items": [
      {"type": "dashboard_by_uid", "value": "<uid1>"},
      {"type": "dashboard_by_uid", "value": "<uid2>"}
    ]
  }'

# Chạy playlist ở kiosk mode (full screen, no nav)
# URL: /playlists/play/<id>?kiosk=tv
```

Quick Reference

```
# Kiểm tra config và rules
promtool check config prometheus.yml
promtool check rules rules/*.yml

# Test alert expression
promtool test rules test.yml

# TSDB stats
curl -s http://localhost:9090/api/v1/status/tsdb | jq '.data'

# Reload config không restart
curl -X POST http://localhost:9090/-/reload
```

```
# Grafana health check
curl http://localhost:3000/api/health
```

```
# test.yml — unit test cho alert rules
rule_files:
  - rules/alerts.yml

evaluation_interval: 1m

tests:
  - interval: 1m
    input_series:
      - series: 'http_requests_total{job="api",status="500"}'
        values: '0 10 20 30 40'
      - series: 'http_requests_total{job="api",status="200"}'
        values: '100 100 100 100 100'
    alert_rule_test:
      - eval_time: 5m
        alertname: HighErrorRate
        exp_alerts:
          - exp_labels:
              severity: critical
              job: api
```

12

GitHub Actions — DevOps Tricks

> Practical reference từ beginner đến advanced. Commands/config bằng tiếng Anh, giải thích bằng tiếng Việt. Tags: B beginner · I intermediate · A advanced · ! gotcha

Workflow Basics

B Trigger events phổ biến

```yaml
on:
  push:
    branches: [main, develop]
    paths:
      - 'src/**'          # Chỉ trigger khi có thay đổi trong src/
      - '!src/**/*.test.ts'  # Loại trừ test files
  pull_request:
    types: [opened, synchronize, reopened]
    branches: [main]
  schedule:
    - cron: '0 2 * * *'  # Mỗi ngày lúc 2am UTC
  workflow_dispatch:       # Cho phép chạy tay từ UI
    inputs:
      environment:
        description: 'Target environment'
        required: true
        default: 'staging'
        type: choice
        options: [staging, production]
```

B Environment variables & secrets

```yaml
env:
  NODE_ENV: production      # Workflow-level env var

jobs:
  deploy:
    env:
      APP_VERSION: ${{ github.sha }}  # Job-level, override workflow-level
    steps:
      - name: Deploy
        env:
          API_KEY: ${{ secrets.API_KEY }}  # Step-level, không log ra
        run: |
          echo "Deploying version $APP_VERSION"
          curl -H "Authorization: $API_KEY" https://api.example.com/deploy
```

! Secrets không được print ra log — GitHub tự redact. Nhưng nếu encode base64 rồi print thì vẫn lộ. Đừng làm vậy.

B Contexts quan trọng

```yaml
steps:
  - run: |
      echo "Repo: ${{ github.repository }}"          # owner/repo
      echo "Branch: ${{ github.ref_name }}"          # main
      echo "SHA: ${{ github.sha }}"                  # full commit SHA
      echo "Actor: ${{ github.actor }}"              # user trigger
```

```
      echo "Run ID: ${{ github.run_id }}"            # unique run ID
      echo "Event: ${{ github.event_name }}"         # push, pull_request...
      echo "PR number: ${{ github.event.pull_request.number }}"
```

Job Configuration

Matrix builds — test nhiều version song song

```yaml
jobs:
  test:
    strategy:
      matrix:
        node: [18, 20, 22]
        os: [ubuntu-latest, macos-latest]
      fail-fast: false    # Không cancel các job khác khi 1 job fail
      max-parallel: 4     # Giới hạn số job chạy đồng thời
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node }}
```

Matrix include/exclude

```yaml
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node: [18, 20]
    include:
      # Thêm config đặc biệt cho 1 combination
      - os: ubuntu-latest
        node: 20
        experimental: true
    exclude:
      # Bỏ qua combination không cần thiết
      - os: windows-latest
        node: 18
```

Job outputs — truyền data giữa các jobs

```yaml
jobs:
  build:
    outputs:
      image-tag: ${{ steps.meta.outputs.tags }}
      version: ${{ steps.version.outputs.value }}
    steps:
      - id: version
        run: echo "value=$(cat VERSION)" >> $GITHUB_OUTPUT
      - id: meta
        uses: docker/metadata-action@v5
        with:
          images: myapp

  deploy:
    needs: build   # Phụ thuộc vào build job
    steps:
      - run: |
          echo "Deploying ${{ needs.build.outputs.image-tag }}"
          echo "Version: ${{ needs.build.outputs.version }}"
```

Concurrency — tránh deploy đè nhau

```yaml
concurrency:
  group: deploy-${{ github.ref }}   # Mỗi branch 1 group riêng
  cancel-in-progress: true          # Hủy run cũ nếu có run mới
```

```yaml
# Cho production: không cancel, chờ run cũ xong
concurrency:
  group: production-deploy
  cancel-in-progress: false
```

I Conditional jobs

```yaml
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - run: npm test

  deploy:
    needs: test
    # Chỉ deploy khi push lên main và test pass
    if: github.ref == 'refs/heads/main' && needs.test.result == 'success'
    steps:
      - run: ./deploy.sh

  notify-failure:
    needs: [test, deploy]
    # Chạy khi bất kỳ job nào fail, kể cả khi job bị cancel
    if: failure()
    steps:
      - run: ./notify-slack.sh "Pipeline failed"
```

Caching

B actions/cache cơ bản

```yaml
- name: Cache node_modules
  uses: actions/cache@v4
  with:
    path: ~/.npm
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
    restore-keys: |
      ${{ runner.os }}-node-
```

I Cache key strategy Cache key quyết định khi nào cache được tái sử dụng. `restore-keys` là fallback — dùng cache cũ nhất khớp prefix.

```yaml
# Strategy tốt: từ specific đến general
key: ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}-${{ hashFiles('requirements-
dev.txt') }}
restore-keys: |
  ${{ runner.os }}-pip-${{ hashFiles('requirements.txt') }}-
  ${{ runner.os }}-pip-
  ${{ runner.os }}-
```

I Cache nhiều thư mục

```yaml
- uses: actions/cache@v4
  with:
    path: |
      ~/.npm
      ~/.cache/Cypress
      node_modules/.cache
    key: ${{ runner.os }}-deps-${{ hashFiles('package-lock.json') }}
```

! Cache size tối đa 10GB/repo. Cache không được dùng trong 7 ngày sẽ bị xóa tự động.

I Setup actions tích hợp cache sẵn

```yaml
# actions/setup-node có cache built-in — dùng thay vì tự cache
- uses: actions/setup-node@v4
  with:
    node-version: 20
    cache: 'npm'          # hoặc 'yarn', 'pnpm'
    cache-dependency-path: '**/package-lock.json'
```

Artifacts

B Upload/download artifacts

```yaml
jobs:
  build:
    steps:
      - run: npm run build
      - uses: actions/upload-artifact@v4
        with:
          name: dist-${{ github.sha }}
          path: dist/
          retention-days: 7    # Mặc định 90 ngày, tốn quota

  deploy:
    needs: build
    steps:
      - uses: actions/download-artifact@v4
        with:
          name: dist-${{ github.sha }}
          path: dist/
      - run: ./deploy.sh dist/
```

I Upload release assets

```yaml
- name: Create Release
  uses: softprops/action-gh-release@v2
  if: startsWith(github.ref, 'refs/tags/')
  with:
    files: |
      dist/*.tar.gz
      dist/*.zip
      checksums.txt
    generate_release_notes: true
    draft: false
    prerelease: ${{ contains(github.ref, '-rc') || contains(github.ref, '-beta') }}
```

Reusable Workflows

I Tạo reusable workflow

```yaml
# .github/workflows/deploy-template.yml
on:
  workflow_call:
    inputs:
      environment:
        required: true
        type: string
      image-tag:
        required: true
        type: string
    secrets:
      DEPLOY_KEY:
        required: true
      KUBECONFIG:
        required: false
```

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: ${{ inputs.environment }}
    steps:
      - run: |
          echo "Deploying ${{ inputs.image-tag }} to ${{ inputs.environment }}"
          # Dùng secret
          echo "${{ secrets.DEPLOY_KEY }}" > deploy.key
```

Gọi reusable workflow

```
# .github/workflows/production.yml
jobs:
  deploy-prod:
    uses: ./.github/workflows/deploy-template.yml
    with:
      environment: production
      image-tag: ${{ needs.build.outputs.tag }}
    secrets:
      DEPLOY_KEY: ${{ secrets.PROD_DEPLOY_KEY }}
      # Hoặc inherit tất cả secrets từ caller
      # secrets: inherit
```

Composite action vs Reusable workflow - **Composite action**: nhóm steps lại, chạy trong context của caller job. Dùng khi muốn tái sử dụng steps. - **Reusable workflow**: chạy như job độc lập, có runner riêng. Dùng khi cần jobs song song hoặc environment riêng.

```
# .github/actions/setup-app/action.yml — Composite action
name: Setup Application
inputs:
  node-version:
    default: '20'
runs:
  using: composite
  steps:
    - uses: actions/setup-node@v4
      with:
        node-version: ${{ inputs.node-version }}
        cache: npm
    - run: npm ci
      shell: bash
    - run: npm run build
      shell: bash
```

```
# Dùng composite action
steps:
  - uses: ./.github/actions/setup-app
    with:
      node-version: '20'
```

Docker in Actions

Build và push Docker image

```
jobs:
  docker:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: docker/setup-buildx-action@v3  # BuildKit cho layer caching

      - uses: docker/login-action@v3
        with:
```

```
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - uses: docker/metadata-action@v5
        id: meta
        with:
          images: ghcr.io/${{ github.repository }}
          tags: |
            type=ref,event=branch
            type=semver,pattern={{version}}
            type=sha,prefix=sha-

      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          cache-from: type=gha          # Cache từ GitHub Actions cache
          cache-to: type=gha,mode=max
          platforms: linux/amd64,linux/arm64
```

I Service containers — test với database

```
jobs:
  test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_DB: testdb
          POSTGRES_USER: test
          POSTGRES_PASSWORD: test
        ports:
          - 5432:5432
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
      redis:
        image: redis:7
        ports:
          - 6379:6379

    env:
      DATABASE_URL: postgres://test:test@localhost:5432/testdb
      REDIS_URL: redis://localhost:6379

    steps:
      - run: npm test
```

Security

B GITHUB_TOKEN permissions — nguyên tắc least privilege

```
# Top-level: mặc định tất cả là read
permissions:
  contents: read
  packages: read

jobs:
  deploy:
    permissions:
      contents: write      # Cần để tạo release
```

```
      packages: write      # Cần để push Docker image
      id-token: write      # Cần cho OIDC
```

⚠ Từ 2023, repo mới mặc định `permissions: read` cho tất cả. Repo cũ có thể vẫn là `write` — nên set explicit.

OIDC — xác thực với cloud không cần long-lived secrets Thay vì lưu AWS/GCP credentials trong secrets, dùng OIDC để lấy temporary credentials.

```yaml
jobs:
  deploy:
    permissions:
      id-token: write   # Bắt buộc cho OIDC
      contents: read

    steps:
      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789:role/github-actions-deploy
          aws-region: ap-southeast-1
          # Không cần AWS_ACCESS_KEY_ID hay AWS_SECRET_ACCESS_KEY

      - run: aws s3 sync dist/ s3://my-bucket/
```

```
// AWS Trust Policy cho IAM Role
{
  "Statement": [{
    "Effect": "Allow",
    "Principal": {"Federated": "arn:aws:iam::123456789:oidc-provider/
token.actions.githubusercontent.com"},
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "token.actions.githubusercontent.com:aud": "sts.amazonaws.com",
        "token.actions.githubusercontent.com:sub": "repo:owner/repo:ref:refs/heads/main"
      }
    }
  }]
}
```

CodeQL — static analysis tích hợp

```yaml
# .github/workflows/codeql.yml
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
  schedule:
    - cron: '30 1 * * 1'   # Quét toàn bộ mỗi thứ 2

jobs:
  analyze:
    runs-on: ubuntu-latest
    permissions:
      security-events: write
      actions: read
      contents: read
    strategy:
      matrix:
        language: [javascript-typescript, python]
    steps:
      - uses: actions/checkout@v4
      - uses: github/codeql-action/init@v3
        with:
          languages: ${{ matrix.language }}
          queries: security-extended   # Thêm các query bảo mật nâng cao
```

```
      - uses: github/codeql-action/autobuild@v3
      - uses: github/codeql-action/analyze@v3
```

Self-Hosted Runners

I **Setup self-hosted runner**

```
# Trên server (Ubuntu)
mkdir actions-runner && cd actions-runner
RUNNER_VERSION=$(curl -s https://api.github.com/repos/actions/runner/releases/latest | jq -r '.tag_name' | sed 's/^v//')
curl -O -L https://github.com/actions/runner/releases/download/v${RUNNER_VERSION}/actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz
tar xzf ./actions-runner-linux-x64-${RUNNER_VERSION}.tar.gz

# Lấy token từ GitHub: Settings > Actions > Runners > New self-hosted runner
./config.sh --url https://github.com/owner/repo --token <TOKEN>

# Chạy như systemd service
sudo ./svc.sh install
sudo ./svc.sh start
```

```
# Dùng self-hosted runner với label
runs-on: [self-hosted, linux, x64, gpu]
```

A **Ephemeral runners với Actions Runner Controller (ARC)** Runner tự tạo/xóa theo demand — không cần quản lý thủ công.

```
# Helm install ARC
helm install arc \
  --namespace arc-systems \
  --create-namespace \
  oci://ghcr.io/actions/actions-runner-controller-charts/gha-runner-scale-set-controller

# Scale set config
helm install arc-runner-set \
  --namespace arc-runners \
  --create-namespace \
  --set githubConfigUrl="https://github.com/owner/repo" \
  --set githubConfigSecret.github_token="<TOKEN>" \
  --set minRunners=0 \
  --set maxRunners=10 \
  oci://ghcr.io/actions/actions-runner-controller-charts/gha-runner-scale-set
```

```
# Workflow dùng ARC runner
runs-on: arc-runner-set
```

Performance

I **Path filters — skip workflow khi không cần thiết**

```
on:
  push:
    paths:
      - 'src/**'
      - 'package*.json'
      - '.github/workflows/**'
    paths-ignore:
      - '**.md'
      - 'docs/**'
      - '.gitignore'
```

! `paths` và `paths-ignore` không thể dùng cùng nhau trong 1 event. Chọn 1 trong 2.

I Skip CI với commit message

```yaml
jobs:
  test:
    if: |
      !contains(github.event.head_commit.message, '[skip ci]') &&
      !contains(github.event.head_commit.message, '[ci skip]')
```

I Timeout để tránh job treo vô hạn

```yaml
jobs:
  build:
    timeout-minutes: 30    # Job timeout

    steps:
      - name: Long running step
        timeout-minutes: 10   # Step timeout
        run: ./build.sh
```

A Parallel test splitting

```yaml
jobs:
  test:
    strategy:
      matrix:
        shard: [1, 2, 3, 4]   # Chia test thành 4 phần
    steps:
      - run: npx playwright test --shard=${{ matrix.shard }}/4
```

Advanced Patterns

I Deployment environments với approval gate

```yaml
jobs:
  deploy-staging:
    runs-on: ubuntu-latest
    environment: staging    # Không cần approval
    steps:
      - run: ./deploy.sh staging

  deploy-production:
    needs: deploy-staging
    runs-on: ubuntu-latest
    environment: production  # Cần approval từ reviewer trong GitHub Settings
    steps:
      - run: ./deploy.sh production
```

A Dynamic matrix — tạo matrix từ output của job trước

```yaml
jobs:
  discover:
    runs-on: ubuntu-latest
    outputs:
      matrix: ${{ steps.find.outputs.matrix }}
    steps:
      - id: find
        run: |
          # Tìm tất cả service có Dockerfile
          SERVICES=$(find . -name Dockerfile -exec dirname {} \; | \
            jq -R . | jq -sc .)
          echo "matrix={\"service\":$SERVICES}" >> $GITHUB_OUTPUT

  build:
```

```
    needs: discover
    strategy:
      matrix: ${{ fromJson(needs.discover.outputs.matrix) }}
    steps:
      - run: docker build ${{ matrix.service }}
```

A Workflow_dispatch với complex inputs

```yaml
on:
  workflow_dispatch:
    inputs:
      services:
        description: 'Services to deploy (JSON array)'
        required: true
        default: '["api","worker","web"]'
      dry-run:
        type: boolean
        default: false

jobs:
  deploy:
    strategy:
      matrix:
        service: ${{ fromJson(inputs.services) }}
    steps:
      - run: |
          if [ "${{ inputs.dry-run }}" = "true" ]; then
            echo "DRY RUN: would deploy ${{ matrix.service }}"
          else
            ./deploy.sh ${{ matrix.service }}
          fi
```

Debugging

B ACTIONS_STEP_DEBUG — verbose logging

```
# Bật debug logging cho 1 run cụ thể
# Vào Actions tab > Re-run jobs > Enable debug logging

# Hoặc set secret trong repo
ACTIONS_STEP_DEBUG = true
ACTIONS_RUNNER_DEBUG = true
```

I SSH vào runner để debug

```yaml
steps:
  - run: npm test
    continue-on-error: true   # Không fail ngay khi test lỗi

  - name: SSH debug session (chỉ dùng khi debug)
    uses: mxschmitt/action-tmate@v3
    if: failure()   # Chỉ mở SSH khi có lỗi
    with:
      limit-access-to-actor: true   # Chỉ cho phép người trigger workflow SSH vào
      timeout-minutes: 30
```

! Nhớ xóa step này sau khi debug xong. Đừng merge PR có tmate step.

I act — test workflow locally

```
# Cài act
brew install act  # macOS
# hoặc
curl https://raw.githubusercontent.com/nektos/act/master/install.sh | sudo bash
```

```
# Chạy workflow locally
act push                          # Simulate push event
act pull_request                  # Simulate PR event
act -j build                      # Chỉ chạy job "build"
act -s GITHUB_TOKEN=$(gh auth token)  # Pass secrets
act --env-file .env.local         # Load env từ file
act --dry-run                     # Xem sẽ chạy gì, không thực thi
```

act dùng Docker để mô phỏng runner. Một số action không hoạt động hoàn hảo locally (OIDC, GitHub-specific services).

Common Recipes

Semantic release tự động

```yaml
# .github/workflows/release.yml
on:
  push:
    branches: [main]

jobs:
  release:
    runs-on: ubuntu-latest
    permissions:
      contents: write
      issues: write
      pull-requests: write
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0          # Cần full history để semantic-release hoạt động
          persist-credentials: false
      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm
      - run: npm ci
      - run: npx semantic-release
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          NPM_TOKEN: ${{ secrets.NPM_TOKEN }}
```

Docker build + push hoàn chỉnh

```yaml
jobs:
  docker:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write
    steps:
      - uses: actions/checkout@v4
      - uses: docker/setup-qemu-action@v3     # Multi-platform support
      - uses: docker/setup-buildx-action@v3
      - uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - uses: docker/metadata-action@v5
        id: meta
        with:
          images: ghcr.io/${{ github.repository }}
      - uses: docker/build-push-action@v6
        with:
          push: ${{ github.event_name != 'pull_request' }}
          tags: ${{ steps.meta.outputs.tags }}
```

```
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Deploy lên Kubernetes

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: ${{ vars.AWS_DEPLOY_ROLE }}
          aws-region: ap-southeast-1

      - run: aws eks update-kubeconfig --name my-cluster --region ap-southeast-1

      - name: Deploy
        run: |
          # Update image tag trong manifest
          sed -i "s|IMAGE_TAG|${{ github.sha }}|g" k8s/deployment.yml
          kubectl apply -f k8s/
          kubectl rollout status deployment/myapp --timeout=5m
```

Terraform plan/apply với PR comment

```yaml
jobs:
  terraform:
    runs-on: ubuntu-latest
    permissions:
      pull-requests: write
      contents: read
      id-token: write
    defaults:
      run:
        working-directory: terraform/

    steps:
      - uses: actions/checkout@v4
      - uses: hashicorp/setup-terraform@v3

      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: ${{ vars.TF_ROLE_ARN }}
          aws-region: ap-southeast-1

      - run: terraform init
      - run: terraform validate

      - id: plan
        run: terraform plan -no-color -out=tfplan
        continue-on-error: true

      # Comment plan output lên PR
      - uses: actions/github-script@v7
        if: github.event_name == 'pull_request'
        with:
          script: |
            const output = `#### Terraform Plan 📋
            \`\`\`
            ${{ steps.plan.outputs.stdout }}
            \`\`\`
            *Result: ${{ steps.plan.outcome }}*`;
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
```

```
              body: output
            })

      - name: Apply (chỉ khi push lên main)
        if: github.ref == 'refs/heads/main' && github.event_name == 'push'
        run: terraform apply tfplan
```

B PR checks — lint, test, build

```yaml
# .github/workflows/pr-checks.yml
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  checks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: npm
      - run: npm ci

      - name: Lint
        run: npm run lint

      - name: Type check
        run: npm run typecheck

      - name: Test
        run: npm test -- --coverage

      - name: Upload coverage
        uses: codecov/codecov-action@v4
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
          fail_ci_if_error: false   # Không fail nếu codecov down

      - name: Build
        run: npm run build
```

Quick Reference

```
# Xem workflow runs
gh run list --workflow=deploy.yml --limit=10

# Xem log của run cụ thể
gh run view <run-id> --log

# Re-run failed jobs
gh run rerun <run-id> --failed

# Download artifact
gh run download <run-id> -n artifact-name

# Trigger workflow_dispatch
gh workflow run deploy.yml -f environment=staging

# Xem secrets (chỉ thấy tên, không thấy value)
gh secret list

# Set secret
gh secret set API_KEY --body "secret-value"
```

```
# Xem environment variables
gh variable list
gh variable set NODE_ENV --body "production"
```

```
# Template: step hay dùng để debug
- name: Debug context
  if: runner.debug == '1'
  run: |
    echo "═══ GitHub Context ═══"
    echo '${{ toJson(github) }}'
    echo "═══ Runner Context ═══"
    echo '${{ toJson(runner) }}'
    echo "═══ Env ═══"
    env | sort
```

13

GitLab CI/CD Tricks — Beginner to Advanced

> Thực chiến. Không lý thuyết thừa. Mỗi trick có ví dụ chạy được. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

1. Pipeline Basics

B Cấu trúc stages và jobs

Stages chạy tuần tự, jobs trong cùng stage chạy song song.

```yaml
stages:
  - build
  - test
  - deploy

build-app:
  stage: build
  script:
    - docker build -t myapp:$CI_COMMIT_SHORT_SHA .

run-tests:
  stage: test
  script:
    - npm test
```

B before_script / after_script

`before_script` chạy trước mọi `script` trong job. `after_script` luôn chạy dù job fail.

```yaml
default:
  before_script:
    - apt-get update -qq && apt-get install -y curl

deploy-job:
  after_script:
    - echo "Cleanup after deploy"   # chạy dù deploy fail
  script:
    - ./deploy.sh
```

B Variables — khai báo và sử dụng

Variables có thể khai báo ở global, job, hoặc UI (Settings > CI/CD > Variables).

```yaml
variables:
  APP_ENV: production
  REGISTRY: registry.gitlab.com/myorg/myapp

build:
  variables:
    DOCKER_BUILDKIT: "1"   # override chỉ trong job này
```

```
  script:
    - docker build -t $REGISTRY:$CI_COMMIT_SHA .
```

rules vs only/except

`only/except` đã deprecated. Dùng `rules` cho logic phức tạp và dễ đọc hơn.

```
# Cũ — tránh dùng
deploy:
  only:
    - main
  except:
    - schedules

# Mới — dùng rules
deploy:
  rules:
    - if: $CI_COMMIT_BRANCH == "main" && $CI_PIPELINE_SOURCE != "schedule"
```

---

2. Rules & Workflow

rules:if — điều kiện chạy job

Dùng biểu thức so sánh với predefined variables.

```
deploy-prod:
  rules:
    - if: $CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/   # chỉ chạy khi tag semver
      when: manual
    - when: never

deploy-staging:
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: on_success
    - when: never
```

rules:changes — chạy khi file thay đổi

Tối ưu pipeline: chỉ build service nào có code thay đổi.

```
build-frontend:
  rules:
    - changes:
        - frontend/**/*
        - package.json
  script:
    - cd frontend && npm run build

build-backend:
  rules:
    - changes:
        - backend/**/*
        - go.sum
  script:
    - go build ./...
```

`rules:changes` so sánh với commit trước. Trên branch mới tạo, so sánh với default branch — có thể gây unexpected behavior.

I rules:exists — chạy khi file tồn tại

Hữu ích cho monorepo — detect xem service có Dockerfile không.

```
build-docker:
  rules:
    - exists:
        - Dockerfile
        - docker/Dockerfile.*
```

I workflow:rules — kiểm soát khi nào tạo pipeline

Ngăn tạo pipeline trùng lặp (push + MR cùng lúc).

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
    - if: $CI_COMMIT_TAG
```

I Merge Request Pipelines

Chạy pipeline riêng cho MR, dùng `$CI_MERGE_REQUEST_*` variables.

```
lint-mr:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  script:
    - echo "Checking MR !$CI_MERGE_REQUEST_IID from $CI_MERGE_REQUEST_SOURCE_BRANCH_NAME"
    - golangci-lint run --new-from-rev=$CI_MERGE_REQUEST_DIFF_BASE_SHA
```

---

3. DAG & Needs

I needs — phá vỡ thứ tự stages

Job có `needs` không cần đợi toàn bộ stage trước, chỉ cần đợi jobs được liệt kê.

```
stages: [build, test, deploy]

build-api:
  stage: build
  script: make build-api

build-ui:
  stage: build
  script: make build-ui

test-api:
  stage: test
  needs: [build-api]      # chạy ngay khi build-api xong, không cần đợi build-ui
  script: make test-api

deploy:
  stage: deploy
  needs: [test-api, build-ui]
  script: make deploy
```

A parallel jobs với matrix

Chạy cùng job với nhiều config khác nhau, tự động tạo job name.

```yaml
test-matrix:
  parallel:
    matrix:
      - PYTHON_VERSION: ["3.10", "3.11", "3.12"]
        DATABASE: ["postgres", "mysql"]
  image: python:$PYTHON_VERSION
  services:
    - name: $DATABASE:latest
  script:
    - pytest tests/

# Tạo ra 6 jobs: test-matrix: [3.10, postgres], [3.10, mysql], ...
```

A needs với optional dependencies

Job có thể không cần fail nếu dependency không tồn tại (do rules).

```yaml
deploy:
  needs:
    - job: build-api
      optional: false
    - job: build-ui
      optional: true   # deploy vẫn chạy nếu build-ui bị skip
  script: make deploy
```

---

4. Caching

B Cache cơ bản — node_modules

```yaml
build:
  cache:
    key: $CI_COMMIT_REF_SLUG      # cache per branch
    paths:
      - node_modules/
  script:
    - npm ci
    - npm run build
```

I Cache key strategies

```yaml
# Cache theo nội dung file lock — cache invalidate khi deps thay đổi
build:
  cache:
    key:
      files:
        - package-lock.json
        - yarn.lock
    paths:
      - node_modules/
      - .npm/

# Cache theo branch + OS
test:
  cache:
    key: "$CI_JOB_NAME-$CI_COMMIT_REF_SLUG"
    paths:
      - .cache/
```

I cache:policy — tách pull và push

Mặc định mỗi job đều pull và push cache. Tách ra để tránh job cuối overwrite cache đúng.

```yaml
build:
  cache:
    key: $CI_COMMIT_REF_SLUG
    paths: [node_modules/]
    policy: push          # chỉ push, không pull (job đầu tiên tạo cache)

test:
  cache:
    key: $CI_COMMIT_REF_SLUG
    paths: [node_modules/]
    policy: pull          # chỉ pull, không push (đọc cache của build)
```

A Distributed caching với S3

Cấu hình runner dùng S3 để share cache giữa nhiều runner instances.

```toml
# /etc/gitlab-runner/config.toml
[runners.cache]
  Type = "s3"
  Path = "gitlab-runner-cache"
  Shared = true
  [runners.cache.s3]
    ServerAddress = "s3.amazonaws.com"
    BucketName = "my-runner-cache"
    BucketLocation = "ap-southeast-1"
    Insecure = false
```

! Cache không được dùng để truyền artifacts giữa jobs — dùng `artifacts` cho việc đó.

---

5. Artifacts

B artifacts:paths — lưu file output

```yaml
build:
  script:
    - npm run build
    - go build -o bin/app ./...
  artifacts:
    paths:
      - dist/
      - bin/app
    expire_in: 1 week
```

I artifacts:reports — reports tích hợp GitLab UI

```yaml
test:
  script:
    - pytest --junitxml=report.xml --cov=. --cov-report=cobertura:coverage.xml
  artifacts:
    reports:
      junit: report.xml
      coverage_report:
        coverage_format: cobertura
        path: coverage.xml
    expire_in: 30 days
```

I dotenv artifacts — pass variables giữa jobs

```yaml
build:
  script:
    - echo "IMAGE_TAG=$CI_COMMIT_SHORT_SHA" >> build.env
```

```
    - echo "BUILD_TIME=$(date -u +%Y%m%dT%H%M%SZ)" >> build.env
  artifacts:
    reports:
      dotenv: build.env

deploy:
  needs:
    - job: build
      artifacts: true
  script:
    - echo "Deploying $IMAGE_TAG built at $BUILD_TIME"
    - kubectl set image deployment/app app=$REGISTRY:$IMAGE_TAG
```

I dependencies — chỉ download artifacts cần thiết

Mặc định job download tất cả artifacts từ stages trước. Dùng `dependencies` để giới hạn.

```
deploy-eu:
  dependencies:
    - build-eu        # chỉ download artifact từ build-eu, bỏ qua build-us
  script: ./deploy.sh eu
```

! `dependencies: []` (list rỗng) = không download artifact nào — khác với không khai báo `dependencies`.

---

6. Includes & Templates

B include:local — tách file config

```
# .gitlab-ci.yml
include:
  - local: '.gitlab/ci/build.yml'
  - local: '.gitlab/ci/deploy.yml'
  - local: '.gitlab/ci/security.yml'
```

I include:template — dùng template có sẵn của GitLab

```
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Secret-Detection.gitlab-ci.yml
  - template: Code-Quality.gitlab-ci.yml
```

I include:project — tái sử dụng config từ project khác

```
include:
  - project: 'myorg/ci-templates'
    ref: main
    file: '/templates/docker-build.yml'
```

A extends — kế thừa job config

```
.base-deploy:
  image: alpine:3.18
  before_script:
    - apk add --no-cache curl
    - curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/
linux/amd64/kubectl"
  environment:
    name: $DEPLOY_ENV

deploy-staging:
```

```
  extends: .base-deploy
  variables:
    DEPLOY_ENV: staging
    KUBE_CONTEXT: staging-cluster

deploy-prod:
  extends: .base-deploy
  variables:
    DEPLOY_ENV: production
  rules:
    - if: $CI_COMMIT_TAG
      when: manual
```

A !reference — reuse phần config cụ thể

```
.setup-aws:
  before_script: &aws-setup
    - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
    - aws configure set region ap-southeast-1

deploy:
  before_script:
    - !reference [.setup-aws, before_script]
    - echo "AWS setup done, deploying..."
  script:
    - aws ecs update-service --cluster prod --service app --force-new-deployment
```

---

7. Environments & Deployments

B environment — tracking deployments

```
deploy-staging:
  environment:
    name: staging
    url: https://staging.myapp.com
  script: ./deploy.sh staging
```

I Dynamic environments — review apps

Tạo môi trường riêng cho từng MR, tự động destroy khi MR merge.

```
review-app:
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: https://$CI_COMMIT_REF_SLUG.review.myapp.com
    on_stop: stop-review-app
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
  script:
    - helm upgrade --install review-$CI_COMMIT_REF_SLUG ./chart
      --set host=$CI_COMMIT_REF_SLUG.review.myapp.com

stop-review-app:
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      when: manual
  script:
    - helm uninstall review-$CI_COMMIT_REF_SLUG
```

A resource_group — ngăn deploy đồng thời

Đảm bảo chỉ một deploy chạy tại một thời điểm cho mỗi môi trường.

```
deploy-prod:
  resource_group: production     # job sau phải chờ job trước xong
  environment: production
  script: ./deploy.sh prod
```

---

8. Docker Integration

B Docker-in-Docker (DinD)

```
build-image:
  image: docker:24.0
  services:
    - docker:24.0-dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  script:
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
```

! DinD cần privileged mode trên runner. Cân nhắc dùng Kaniko thay thế cho môi trường không cho phép privileged.

I Kaniko — build không cần privileged

```
build-kaniko:
  image:
    name: gcr.io/kaniko-project/executor:v1.21.0-debug
    entrypoint: [""]
  script:
    - /kaniko/executor
      --context $CI_PROJECT_DIR
      --dockerfile $CI_PROJECT_DIR/Dockerfile
      --destination $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
      --cache=true
      --cache-repo=$CI_REGISTRY_IMAGE/cache
```

A Multi-arch builds với buildx

```
build-multiarch:
  image: docker:24.0
  services:
    - docker:24.0-dind
  before_script:
    - docker run --privileged --rm tonistiigi/binfmt --install all
    - docker buildx create --use --name multiarch
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
  script:
    - docker buildx build
      --platform linux/amd64,linux/arm64
      --tag $CI_REGISTRY_IMAGE:$CI_COMMIT_TAG
      --push .
```

---

9. Runners

B Runner tags — điều hướng jobs

```yaml
# Runner được cấu hình với tags: [gpu, large]
train-model:
  tags:
    - gpu
    - large
  script:
    - python train.py

# Job không có tags chạy trên shared runners
lint:
  script:
    - golangci-lint run
```

I Runner autoscaling với Docker Machine

```
# config.toml — autoscaler tạo VM theo nhu cầu
[runners.machine]
  IdleCount = 1
  IdleTime = 1800
  MaxBuilds = 10
  MachineDriver = "amazonec2"
  MachineName = "gitlab-runner-%s"
  MachineOptions = [
    "amazonec2-instance-type=t3.medium",
    "amazonec2-region=ap-southeast-1",
    "amazonec2-ami=ami-0c55b159cbfafe1f0"
  ]
```

A Runner concurrent và job timeout

```
# config.toml
concurrent = 10          # chạy tối đa 10 jobs đồng thời trên runner này

[[runners]]
  name = "production-runner"
  limit = 5              # runner này nhận tối đa 5 jobs
  output_limit = 4096    # giới hạn log output (KB)

  [runners.custom_build_dir]
    enabled = true
```

10. Security Scanning

I SAST + Secret Detection

```yaml
include:
  - template: Security/SAST.gitlab-ci.yml
  - template: Security/Secret-Detection.gitlab-ci.yml
  - template: Security/Dependency-Scanning.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml

# Customize SAST
sast:
  variables:
    SAST_EXCLUDED_PATHS: "spec, test, tests, tmp"
    SEARCH_MAX_DEPTH: 10
```

```yaml
# Container scanning cần image được build trước
container_scanning:
  needs: [build-image]
  variables:
    CS_IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
```

A Compliance Framework — enforce security jobs

Policy được enforce ở group level, không thể bị override bởi project.

```yaml
# Trong compliance pipeline project (group level)
include:
  - template: Security/SAST.gitlab-ci.yml

# Project không thể tắt SAST nếu compliance framework được áp dụng
```

---

11. Variables

B Predefined variables hữu ích nhất

```yaml
$CI_COMMIT_SHA           # full SHA
$CI_COMMIT_SHORT_SHA     # 8 ký tự đầu
$CI_COMMIT_BRANCH        # tên branch
$CI_COMMIT_TAG           # tag name (nếu có)
$CI_COMMIT_REF_SLUG      # branch/tag dưới dạng slug (lowercase, hyphen)
$CI_PROJECT_NAME         # tên project
$CI_PROJECT_PATH         # group/project-name
$CI_REGISTRY_IMAGE       # đường dẫn registry mặc định
$CI_PIPELINE_SOURCE      # push, merge_request_event, schedule, trigger, ...
$CI_MERGE_REQUEST_IID    # MR number
$CI_ENVIRONMENT_SLUG     # environment name dạng slug
```

I Protected và Masked variables

- **Protected**: chỉ expose trên protected branches/tags
- **Masked**: không hiển thị trong job logs

```yaml
# Trong job, biến masked sẽ hiện là [MASKED]
deploy:
  script:
    - echo $AWS_SECRET_KEY   # output: [MASKED]
    - aws s3 sync dist/ s3://my-bucket
```

A Variable inheritance — thứ tự ưu tiên

Từ thấp đến cao (cao hơn override thấp hơn): 1. Predefined variables 2. Group-level CI/CD variables 3. Project-level CI/CD variables 4. `.gitlab-ci.yml` global `variables:` 5. Job-level `variables:` 6. Trigger variables (pipeline trigger)

! Biến được set trong `before_script` hoặc `script` dùng `export` không tự động truyền sang job tiếp theo — phải dùng dotenv artifacts.

---

12. Advanced Pipeline Patterns

I Parent-child pipelines — tách pipeline lớn

```yaml
# parent .gitlab-ci.yml
trigger-api:
  trigger:
    include: services/api/.gitlab-ci.yml
    strategy: depend     # parent fail nếu child fail
  rules:
    - changes:
        - services/api/**

trigger-ui:
  trigger:
    include: services/ui/.gitlab-ci.yml
  rules:
    - changes:
        - services/ui/**
```

I Multi-project pipelines — trigger pipeline dự án khác

```yaml
# Project A trigger deploy pipeline của Project B
trigger-deploy:
  trigger:
    project: myorg/deployment-repo
    branch: main
    strategy: depend
  variables:
    APP_VERSION: $CI_COMMIT_TAG
    TRIGGERED_BY: $CI_PROJECT_NAME
```

A interruptible — hủy pipeline cũ khi có pipeline mới

```yaml
# Áp dụng cho toàn bộ pipeline
default:
  interruptible: true

# Override cho deploy job — không hủy khi đang deploy
deploy-prod:
  interruptible: false
  script: ./deploy.sh
```

13. Performance

B retry — tự động retry khi fail

```yaml
flaky-test:
  retry:
    max: 2
    when:
      - runner_system_failure
      - stuck_or_timeout_failure
      - api_failure
```

I timeout — giới hạn thời gian chạy

```yaml
long-build:
  timeout: 2 hours 30 minutes   # override project default timeout
```

```
quick-lint:
  timeout: 5 minutes
```

I parallel — chia nhỏ test suite

```
test:
  parallel: 5    # tạo 5 jobs, mỗi job nhận $CI_NODE_INDEX và $CI_NODE_TOTAL
  script:
    - bundle exec rspec $(ruby scripts/split_tests.rb)
    # scripts/split_tests.rb dùng CI_NODE_INDEX/CI_NODE_TOTAL để chia files
```

A Job log sections — collapsible logs

```
#!/bin/bash
start_section() {
  echo -e "\e[0Ksection_start:$(date +%s):$1\r\e[0K\e[34;1m$2\e[0m"
}
end_section() {
  echo -e "\e[0Ksection_end:$(date +%s):$1\r\e[0K"
}

start_section "install_deps" "Installing dependencies"
npm ci
end_section "install_deps"
```

---

14. Debugging

B Pipeline lint — validate config trước khi push

```
# Lint qua API (không cần đăng nhập)
curl -s --header "PRIVATE-TOKEN: $GITLAB_TOKEN" \
  "https://gitlab.com/api/v4/projects/$PROJECT_ID/ci/lint" \
  --form "content=@.gitlab-ci.yml" | jq '.errors, .warnings'

# Dùng gitlab-runner local
gitlab-runner exec docker build-job
```

I CI_DEBUG_TRACE — log chi tiết mọi command

```
debug-job:
  variables:
    CI_DEBUG_TRACE: "true"    # log tất cả bash commands và env vars
  script:
    - ./complex-script.sh
```

! `CI_DEBUG_TRACE` sẽ log tất cả biến môi trường kể cả secrets — chỉ dùng tạm thời, không commit.

I Pipeline visualization và editor

- GitLab UI > CI/CD > Pipelines > Pipeline graph: xem DAG visualization
- GitLab UI > CI/CD > Editor: lint và visualize trực tiếp trên browser
- Settings > CI/CD > Variables: debug variable inheritance

A Replay job với SSH debug access

```
# Thêm vào job cần debug
debug-session:
```

```
  script:
    - curl -s https://packagecloud.io/install/repositories/tmate/tmate/script.deb.sh | bash
    - apt-get install -y tmate
    - tmate -S /tmp/tmate.sock new-session -d
    - tmate -S /tmp/tmate.sock wait tmate-ready
    - tmate -S /tmp/tmate.sock display -p '#{tmate_ssh}'   # in SSH command ra log
    - sleep 1800   # 30 phút để debug
```

Quick Reference

Trick	Dùng khi nào
`needs:`	Job phụ thuộc vào job cụ thể, không phải toàn stage
`rules:changes:`	Monorepo — chỉ build service có thay đổi
`cache:policy: pull`	Read-only jobs không cần update cache
`artifacts:reports:dotenv:`	Pass dynamic values giữa jobs
`resource_group:`	Serialize deploys, tránh race condition
`interruptible: true`	Feature branch builds — hủy build cũ khi push mới
`parallel:matrix:`	Test nhiều version/config mà không viết lặp
`trigger:strategy:depend`	Parent pipeline cần biết child fail/pass

14

Networking & DNS Tricks — DevOps từ Beginner đến Advanced

> Thực chiến. Không lý thuyết thừa. Mỗi trick có ví dụ chạy được. Tags: B Beginner · I Intermediate · A Advanced · ! Gotcha

1. TCP/IP Fundamentals

B Three-way handshake — hiểu để debug connection issues

```
Client ──SYN──────────────► Server
Client ◄──SYN-ACK──────────── Server
Client ──ACK──────────────► Server   ← connection established
```

```
# Xem trạng thái connections
ss -tan | grep ESTABLISHED | wc -l    # đếm connections đang mở
ss -tan state syn-sent                 # connections đang chờ SYN-ACK (client side)
ss -tan state syn-recv                 # connections đang chờ hoàn thành handshake (server)
```

I TIME_WAIT — nguyên nhân "too many open connections"

Sau khi close connection, socket ở TIME_WAIT 2×MSL (~60-120s) để đảm bảo không có packet cũ lạc.

```
ss -tan | awk '{print $1}' | sort | uniq -c | sort -rn
# Xem phân bố trạng thái. TIME_WAIT nhiều = load balancer hoặc reverse proxy đóng connections

# Giảm TIME_WAIT timeout (production — cẩn thận)
sysctl -w net.ipv4.tcp_fin_timeout=30
sysctl -w net.ipv4.tcp_tw_reuse=1      # cho phép reuse TIME_WAIT sockets cho outgoing
```

! `net.ipv4.tcp_tw_recycle` đã bị xóa khỏi kernel 4.12+. Đừng set nó — kernel sẽ ignore hoặc panic.

I Connection backlog — khi server không kịp accept

```
# Xem backlog queue bị overflow
netstat -s | grep -i "listen"
# "X SYNs to LISTEN sockets dropped" = backlog overflow

# Tăng backlog
sysctl -w net.core.somaxconn=65535          # max backlog per socket
sysctl -w net.ipv4.tcp_max_syn_backlog=8192 # max SYN_RECV queue

# Trong app code (ví dụ Go):
# net.Listen("tcp", ":8080") → listener backlog mặc định = somaxconn
```

A Socket tuning — high performance server

```
# /etc/sysctl.d/99-network-tuning.conf
net.core.rmem_max = 134217728          # receive buffer max 128MB
net.core.wmem_max = 134217728          # send buffer max 128MB
net.ipv4.tcp_rmem = 4096 87380 134217728
net.ipv4.tcp_wmem = 4096 65536 134217728
net.core.netdev_max_backlog = 30000
net.ipv4.tcp_slow_start_after_idle = 0  # không reset congestion window khi idle
```

---

2. DNS

B dig — swiss army knife của DNS

```
dig google.com                      # query mặc định (A record)
dig google.com MX                   # MX records
dig google.com ANY                  # tất cả records (thường bị từ chối)
dig +short google.com               # chỉ in IP, bỏ boilerplate
dig @8.8.8.8 google.com             # query qua DNS cụ thể (bypass /etc/resolv.conf)
dig +trace google.com               # trace toàn bộ resolution chain từ root
dig -x 8.8.8.8                      # reverse lookup (PTR)
```

B Các loại DNS records cần nhớ

```
A       → IPv4 address
AAAA    → IPv6 address
CNAME   → alias trỏ đến domain khác (không dùng cho root domain)
MX      → mail server (kèm priority)
TXT     → text data (SPF, DKIM, verification)
NS      → nameserver của domain
PTR     → reverse DNS (IP → domain)
SRV     → service discovery (host, port, priority, weight)
CAA     → chỉ định CA được phép issue cert
```

I DNS caching — debug stale records

```
# Xem TTL còn lại (TTL giảm dần mỗi lần query nếu có caching)
dig +nocmd +noall +answer +ttlid google.com

# Flush DNS cache
systemd-resolve --flush-caches        # systemd-resolved (Ubuntu 18.04+)
resolvectl flush-caches               # alias mới hơn
nscd -i hosts                         # nscd

# Xem cache stats
resolvectl statistics
systemd-resolve --statistics
```

I Split-horizon DNS — internal vs external resolution

```
# /etc/hosts override (local machine)
192.168.1.100  api.myapp.com   # force resolve về internal IP

# Dnsmasq — split-horizon cho toàn network
# /etc/dnsmasq.conf
address=/internal.myapp.com/10.0.0.5    # trả về IP này cho domain này
server=/myapp.com/10.0.0.1               # forward .myapp.com queries đến internal DNS

# Systemd-resolved — per-interface DNS
# /etc/systemd/resolved.conf.d/vpn.conf
[Resolve]
```

```
DNS=10.0.0.1
Domains=~internal.myapp.com    # tilde = routing domain, không phải search domain
```

A DNS debugging nâng cao

```
# Xem DNS resolution chain thực tế (kể cả /etc/hosts, nsswitch)
getent hosts myapp.com             # dùng NSS stack, giống app thực tế

# Capture DNS traffic
tcpdump -i any -n port 53

# Kiểm tra DNSSEC
dig +dnssec cloudflare.com
dig +short DS cloudflare.com @8.8.8.8

# So sánh response từ authoritative vs recursive
dig @ns1.cloudflare.com cloudflare.com     # trực tiếp từ authoritative NS
dig @1.1.1.1 cloudflare.com               # từ recursive resolver
```

---

3. TLS/SSL

B openssl — kiểm tra certificate

```
# Xem cert của domain (bao gồm chain)
openssl s_client -connect google.com:443 -servername google.com < /dev/null

# Xem thông tin cert
openssl s_client -connect google.com:443 < /dev/null 2>/dev/null \
  | openssl x509 -noout -text | grep -A2 "Subject:\|Issuer:\|Not After"

# Kiểm tra cert file local
openssl x509 -in cert.pem -noout -dates -subject -issuer

# Kiểm tra cert khớp với private key
openssl x509 -noout -modulus -in cert.pem | md5sum
openssl rsa  -noout -modulus -in key.pem  | md5sum
# Hai hash phải giống nhau
```

I Certificate chain debugging

```
# Verify chain hoàn chỉnh
openssl verify -CAfile /etc/ssl/certs/ca-certificates.crt cert.pem

# Download và kiểm tra toàn bộ chain
openssl s_client -connect myapp.com:443 -showcerts < /dev/null 2>/dev/null \
  | awk '/-----BEGIN CERTIFICATE-----/,/-----END CERTIFICATE-----/' \
  | csplit - '/-----BEGIN CERTIFICATE-----/' '{*}'
# Tạo ra các file cert riêng để verify

# Xem cert expiry của nhiều domain
for domain in google.com github.com cloudflare.com; do
  exp=$(echo | openssl s_client -servername $domain -connect $domain:443 2>/dev/null \
    | openssl x509 -noout -enddate 2>/dev/null | cut -d= -f2)
  echo "$domain: $exp"
done
```

I mTLS — mutual TLS, cả client và server đều authenticate

```
# Tạo CA tự ký
openssl req -new -x509 -days 3650 -nodes -out ca.crt -keyout ca.key \
  -subj "/CN=My Internal CA"
```

```
# Tạo server cert
openssl req -new -nodes -out server.csr -keyout server.key -subj "/CN=myapp.internal"
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
  -out server.crt -days 365

# Tạo client cert
openssl req -new -nodes -out client.csr -keyout client.key -subj "/CN=my-service"
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial \
  -out client.crt -days 365

# Test với curl
curl --cert client.crt --key client.key --cacert ca.crt https://myapp.internal/
```

A ACME/Let's Encrypt internals

```
# Certbot standalone (tạm thời mở port 80)
certbot certonly --standalone -d myapp.com

# DNS-01 challenge (không cần port 80 — dùng cho wildcard)
certbot certonly --manual --preferred-challenges dns -d "*.myapp.com"

# Cloudflare DNS plugin — tự động
pip install certbot-dns-cloudflare
certbot certonly --dns-cloudflare \
  --dns-cloudflare-credentials ~/.secrets/cloudflare.ini \
  -d "*.myapp.com" -d "myapp.com"

# Xem cert renewal schedule
certbot renew --dry-run
systemctl list-timers | grep certbot
```

---

4. Linux Networking

B ip command — thay thế ifconfig/route

```
ip addr show                    # xem tất cả interfaces và IPs
ip addr show eth0               # chỉ eth0
ip link set eth0 up/down        # bật/tắt interface
ip route show                   # routing table
ip route get 8.8.8.8            # xem route đến địa chỉ cụ thể
ip neigh show                   # ARP table
ip -s link                      # statistics (packets, errors)
```

I Routing tables và policy routing

```
# Thêm route tĩnh
ip route add 10.0.0.0/8 via 192.168.1.1 dev eth0

# Route với metric (ưu tiên thấp hơn)
ip route add default via 192.168.1.1 metric 100

# Policy routing — route theo source IP
ip rule add from 10.0.1.0/24 table 100   # traffic từ subnet này dùng table 100
ip route add default via 10.0.1.1 table 100

# Xem tất cả routing tables
ip rule show
ip route show table all
```

I Network namespaces — isolation cho containers

```
# Tạo namespace
ip netns add myns

# Chạy command trong namespace
ip netns exec myns ip addr show
ip netns exec myns ping 8.8.8.8

# Tạo veth pair (pipe hai chiều giữa namespaces)
ip link add veth0 type veth peer name veth1
ip link set veth1 netns myns
ip addr add 10.0.0.1/30 dev veth0
ip netns exec myns ip addr add 10.0.0.2/30 dev veth1
ip link set veth0 up
ip netns exec myns ip link set veth1 up

# Xem namespaces
ip netns list
ls /var/run/netns/
```

A Bridge và VLAN

```
# Tạo bridge (software switch)
ip link add br0 type bridge
ip link set eth1 master br0
ip link set eth2 master br0
ip link set br0 up

# VLAN tagging trên interface
ip link add link eth0 name eth0.100 type vlan id 100
ip addr add 192.168.100.1/24 dev eth0.100
ip link set eth0.100 up

# Bridge với VLAN filtering
ip link add br0 type bridge vlan_filtering 1
bridge vlan add dev eth1 vid 100
bridge vlan add dev eth2 vid 100
```

---

5. iptables/nftables

B iptables — xem và debug rules

```
iptables -L -n -v                    # xem tất cả rules với stats
iptables -L INPUT -n -v --line-numbers # INPUT chain với số thứ tự
iptables -t nat -L -n -v             # NAT table
iptables -t mangle -L -n -v          # mangle table

# Xem connection tracking
conntrack -L                          # tất cả tracked connections
conntrack -L | grep ESTABLISHED | wc -l
```

I Các rules phổ biến

```
# Cho phép SSH (tránh tự lock out trước khi thêm rules khác)
iptables -A INPUT -p tcp --dport 22 -j ACCEPT

# Cho phép established connections trở về
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT

# Port forwarding (DNAT)
iptables -t nat -A PREROUTING -p tcp --dport 8080 -j DNAT --to-destination 10.0.0.5:80
```

```
# Masquerade (SNAT cho outgoing — NAT giống router gia đình)
iptables -t nat -A POSTROUTING -s 10.0.0.0/24 -o eth0 -j MASQUERADE

# Rate limiting (chống brute force)
iptables -A INPUT -p tcp --dport 22 -m recent --set --name ssh
iptables -A INPUT -p tcp --dport 22 -m recent --update --seconds 60 \
  --hitcount 4 --name ssh -j DROP

# Lưu rules
iptables-save > /etc/iptables/rules.v4
```

A nftables — thay thế iptables hiện đại

```
# Xem ruleset
nft list ruleset

# Tạo table và chain
nft add table inet filter
nft add chain inet filter input { type filter hook input priority 0 \; policy drop \; }

# Thêm rules
nft add rule inet filter input ct state established,related accept
nft add rule inet filter input iif lo accept
nft add rule inet filter input tcp dport 22 accept
nft add rule inet filter input tcp dport { 80, 443 } accept

# NAT với nftables
nft add table nat
nft add chain nat prerouting { type nat hook prerouting priority -100 \; }
nft add chain nat postrouting { type nat hook postrouting priority 100 \; }
nft add rule nat postrouting oif eth0 masquerade

# File config
cat /etc/nftables.conf
```

---

6. Load Balancing Concepts

B L4 vs L7 Load Balancing

```
L4 (TCP/UDP):
  - Forward packets theo IP:port
  - Không đọc nội dung packet
  - Nhanh hơn, ít CPU hơn
  - Ví dụ: HAProxy TCP mode, AWS NLB

L7 (HTTP/HTTPS):
  - Đọc HTTP headers, path, cookies
  - Routing theo URL path, header
  - SSL termination, compression, caching
  - Ví dụ: nginx, HAProxy HTTP mode, AWS ALB
```

I Thuật toán load balancing

```
# nginx — Round Robin (mặc định)
upstream backend {
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
}

# Least connections
upstream backend {
    least_conn;
```

```
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
}

# IP Hash (sticky sessions theo IP)
upstream backend {
    ip_hash;
    server 10.0.0.1:8080;
    server 10.0.0.2:8080;
}

# Weighted
upstream backend {
    server 10.0.0.1:8080 weight=3;   # nhận 75% traffic
    server 10.0.0.2:8080 weight=1;   # nhận 25% traffic
}
```

A Connection draining — graceful shutdown

```
# nginx — mark server down nhưng giữ existing connections
# upstream backend {
#     server 10.0.0.1:8080;
#     server 10.0.0.2:8080 down;   # không gửi request mới, nhưng existing vẫn chạy
# }

# HAProxy — graceful remove server
echo "disable server backend/server2" | socat stdio /var/run/haproxy/admin.sock
# Sau khi connections drain xong:
echo "set server backend/server2 state drain" | socat stdio /var/run/haproxy/admin.sock
```

---

7. WireGuard

B Setup WireGuard cơ bản

```
# Tạo key pair
wg genkey | tee server_private.key | wg pubkey > server_public.key
wg genkey | tee client_private.key | wg pubkey > client_public.key

# Server config: /etc/wireguard/wg0.conf
[Interface]
Address = 10.8.0.1/24
ListenPort = 51820
PrivateKey = <server_private_key>
PostUp = iptables -A FORWARD -i wg0 -j ACCEPT; iptables -t nat -A POSTROUTING -o eth0 -j MASQUERA
DE
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT; iptables -t nat -D POSTROUTING -o eth0 -j MASQUE
RADE

[Peer]
PublicKey = <client_public_key>
AllowedIPs = 10.8.0.2/32   # IP của client trong VPN

# Bật
wg-quick up wg0
systemctl enable wg-quick@wg0
```

I Hub-and-spoke topology — nhiều clients qua một server

```
# Client config: /etc/wireguard/wg0.conf
[Interface]
Address = 10.8.0.2/32
PrivateKey = <client_private_key>
DNS = 10.8.0.1                    # dùng DNS của server
```

```
[Peer]
PublicKey = <server_public_key>
Endpoint = server.myapp.com:51820
AllowedIPs = 0.0.0.0/0             # route ALL traffic qua VPN (full tunnel)
# AllowedIPs = 10.8.0.0/24         # chỉ route VPN subnet (split tunnel)
PersistentKeepalive = 25           # giữ kết nối qua NAT
```

A Site-to-site WireGuard — kết nối hai mạng

```
# Site A server (10.0.1.0/24 LAN)
[Interface]
Address = 10.8.0.1/30
PrivateKey = <site_a_private>
ListenPort = 51820
PostUp = ip route add 10.0.2.0/24 via 10.8.0.2   # route đến Site B LAN

[Peer]
PublicKey = <site_b_public>
Endpoint = site-b.myapp.com:51820
AllowedIPs = 10.8.0.2/32, 10.0.2.0/24  # Site B VPN IP + Site B LAN
PersistentKeepalive = 25
```

! AllowedIPs trong WireGuard là cả routing table lẫn firewall — packet từ IP không có trong AllowedIPs sẽ bị drop ngay tại kernel.

---

8. SSH Tunneling

B Local port forwarding — truy cập remote service qua SSH

```
# Truy cập database ở remote (không expose port ra ngoài)
ssh -L 5432:localhost:5432 user@bastion-host
# Sau đó: psql -h localhost -p 5432 -U myuser mydb

# Forward qua jump host đến server thứ 3
ssh -L 8080:internal-server:80 user@bastion-host
```

B Remote port forwarding — expose local service ra internet

```
# Expose localhost:3000 thành remote:8080
ssh -R 8080:localhost:3000 user@remote-server
# Người dùng remote-server có thể curl localhost:8080 → đến localhost:3000 của bạn

# Public-facing (thêm GatewayPorts yes vào /etc/ssh/sshd_config trên remote)
ssh -R 0.0.0.0:8080:localhost:3000 user@remote-server
```

I Dynamic SOCKS proxy — dùng SSH như proxy server

```
# Tạo SOCKS5 proxy trên localhost:1080 qua SSH
ssh -D 1080 -N user@remote-server

# Dùng với curl
curl --socks5 localhost:1080 https://internal-service.com

# Dùng với browser (Firefox: Network Settings → Manual proxy → SOCKS5 localhost:1080)
```

I ProxyJump — nhảy qua nhiều bastion hosts

```
# Một lần jump
ssh -J bastion-user@bastion.myapp.com target-user@internal-server

# ~/.ssh/config — cấu hình lâu dài
Host internal-*
    ProxyJump bastion
    User myuser

Host bastion
    HostName bastion.myapp.com
    User bastion-user
    IdentityFile ~/.ssh/bastion_key

# Sau đó chỉ cần:
ssh internal-db-server
```

A SSH multiplexing — reuse connection, tránh handshake lặp

```
# ~/.ssh/config
Host *
    ControlMaster auto
    ControlPath ~/.ssh/sockets/%r@%h-%p
    ControlPersist 10m    # giữ socket 10 phút sau khi session đóng

mkdir -p ~/.ssh/sockets

# Lần SSH đầu: tạo master connection
# Lần SSH sau đến cùng host: reuse socket ngay lập tức (< 100ms)

# Kiểm tra socket
ssh -O check user@remote-server

# Đóng master connection
ssh -O exit user@remote-server
```

---

9. HTTP

B HTTP/2 vs HTTP/3

```
HTTP/1.1: 1 request/connection, head-of-line blocking
HTTP/2:   multiplexing nhiều streams trên 1 TCP connection, header compression (HPACK)
HTTP/3:   dùng QUIC (UDP), không có TCP head-of-line blocking, 0-RTT reconnect

# Kiểm tra protocol đang dùng
curl -I --http2 https://google.com | grep -i "^HTTP"
curl -I --http3 https://cloudflare.com    # nếu curl có HTTP/3 support
```

I Connection reuse và keep-alive

```
# Kiểm tra keep-alive response headers
curl -v https://myapp.com 2>&1 | grep -i "keep-alive\|connection"

# curl — đo thời gian với connection reuse
curl -w "time_connect: %{time_connect}\ntime_appconnect: %{time_appconnect}\n" \
     -o /dev/null -s https://myapp.com

# nginx — tuning keepalive
# upstream keepalive (HTTP/1.1 với backend)
upstream backend {
    server 10.0.0.1:8080;
```

```
    keepalive 32;          # pool 32 persistent connections đến backend
}
server {
    location / {
        proxy_http_version 1.1;
        proxy_set_header Connection "";   # quan trọng — clear "Connection: close" header
        proxy_pass http://backend;
    }
}
```

A WebSocket và gRPC qua nginx

```
# WebSocket proxy
location /ws {
    proxy_pass http://backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 3600s;    # giữ connection lâu
}

# gRPC proxy (HTTP/2)
location /grpc {
    grpc_pass grpc://10.0.0.1:50051;
    # hoặc với TLS: grpcs://10.0.0.1:50051
}
```

---

10. Debugging Tools

B tcpdump — capture traffic

```
# Capture traffic trên interface
tcpdump -i eth0                          # tất cả traffic
tcpdump -i eth0 port 443                 # chỉ HTTPS
tcpdump -i eth0 host 8.8.8.8             # đến/từ IP cụ thể
tcpdump -i any port 53                   # DNS trên mọi interface
tcpdump -i eth0 'tcp[tcpflags] & tcp-syn != 0'   # chỉ SYN packets

# Lưu ra file để mở trong Wireshark
tcpdump -i eth0 -w capture.pcap

# Đọc file capture
tcpdump -r capture.pcap -n

# Verbose — xem payload (HTTP text)
tcpdump -i eth0 port 80 -A | grep "GET\|POST\|HTTP"
```

B curl verbose — debug HTTP requests

```
# Xem toàn bộ request/response headers
curl -v https://myapp.com

# Đo thời gian từng giai đoạn
curl -w "\n
DNS lookup:    %{time_namelookup}s
TCP connect:   %{time_connect}s
TLS handshake: %{time_appconnect}s
TTFB:          %{time_starttransfer}s
Total:         %{time_total}s
\n" -o /dev/null -s https://myapp.com

# Gửi request với custom headers và body
curl -X POST https://api.myapp.com/users \
```

```
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"name":"test"}' \
  -v 2>&1 | tee debug.log
```

I mtr — traceroute + ping kết hợp, real-time

```
mtr google.com             # interactive mode
mtr --report google.com    # report mode (100 packets rồi in)
mtr --tcp --port 443 google.com   # TCP trace thay vì ICMP (bypass firewall chặn ICMP)
mtr --no-dns google.com           # không resolve DNS, nhanh hơn
```

I nc/ncat — swiss army knife network

```
# Test port connectivity
nc -zv myserver.com 80          # -z = scan only, -v = verbose
nc -zv myserver.com 80-100      # scan range

# Simple HTTP request
echo -e "GET / HTTP/1.0\r\nHost: myapp.com\r\n\r\n" | nc myapp.com 80

# Lắng nghe port (debugging inbound connections)
nc -l 8080

# Transfer file
# Server: nc -l 9000 > received.tar.gz
# Client: nc server 9000 < file.tar.gz

# UDP test
nc -u -z 8.8.8.8 53   # test UDP port 53
```

A ss — thay thế netstat, nhanh hơn

```
ss -tulpn                    # tất cả listening ports + process
ss -tan                      # TCP all states, numeric
ss -o state established      # chỉ ESTABLISHED
ss -s                        # summary statistics
ss 'sport = :8080'           # filter theo source port
ss 'dst 10.0.0.1'            # filter theo destination IP
ss -p                        # hiện process name/PID
```

---

11. Container Networking

B Docker networking modes

```
# Bridge (mặc định) — container có private IP, NAT ra ngoài
docker run --network bridge nginx

# Host — container dùng network stack của host, không NAT
docker run --network host nginx

# None — không có network
docker run --network none busybox

# Xem network
docker network ls
docker network inspect bridge
docker inspect container_name | jq '.[].NetworkSettings'
```

I Docker overlay — multi-host networking (Swarm)

```
# Tạo overlay network
docker network create --driver overlay --attachable my-overlay

# Containers trên các host khác nhau có thể communicate
# qua overlay (VXLAN tunneling trên UDP port 4789)

# Debug overlay
docker network inspect my-overlay
# Xem VXLAN traffic
tcpdump -i eth0 port 4789
```

A Kubernetes networking model

```
Nguyên tắc K8s networking:
1. Mọi Pod có IP riêng
2. Pod-to-Pod communication không qua NAT
3. Node-to-Pod communication không qua NAT
4. Pod thấy chính IP của nó (không phải IP sau NAT)

CNI plugins phổ biến:
- Flannel: đơn giản, VXLAN, phù hợp development
- Calico: policy enforcement mạnh, BGP routing, production
- Cilium: eBPF-based, L7 policy, observability tốt

# Debug pod networking
kubectl exec -it pod-name -- ip addr
kubectl exec -it pod-name -- ip route
kubectl exec -it pod-name -- nslookup kubernetes.default
```

```
# Xem Services và Endpoints
kubectl get endpoints my-service
kubectl describe svc my-service

# Debug DNS trong cluster
kubectl run tmp --image=busybox --rm -it -- nslookup my-service.default.svc.cluster.local

# Xem kube-proxy rules (iptables mode)
iptables -t nat -L KUBE-SERVICES -n --line-numbers
```

---

12. Performance Tuning

I TCP BBR — congestion control cho WAN

```
# Kiểm tra congestion control hiện tại
sysctl net.ipv4.tcp_congestion_control

# Bật BBR (kernel 4.9+)
sysctl -w net.ipv4.tcp_congestion_control=bbr
sysctl -w net.core.default_qdisc=fq

# Lưu vĩnh viễn
echo "net.core.default_qdisc=fq" >> /etc/sysctl.conf
echo "net.ipv4.tcp_congestion_control=bbr" >> /etc/sysctl.conf

# Verify
sysctl net.ipv4.tcp_congestion_control    # phải ra "bbr"
ss -ti | grep bbr                         # xem connections đang dùng BBR
```

I SO_REUSEPORT — phân phối load qua nhiều processes

```
# Cho phép nhiều sockets bind cùng một port
# Kernel tự load balance incoming connections

# Kiểm tra nếu app đang dùng
ss -tlnp | grep :8080
# Multiple processes cùng listen :8080 → SO_REUSEPORT đang hoạt động

# nginx dùng SO_REUSEPORT mặc định (worker_processes > 1)
# /etc/nginx/nginx.conf
# worker_processes auto;
```

A Network buffer tuning — throughput cao

```
# /etc/sysctl.d/99-net-performance.conf
net.core.rmem_default = 1048576
net.core.rmem_max = 134217728
net.core.wmem_default = 1048576
net.core.wmem_max = 134217728
net.ipv4.tcp_rmem = 4096 1048576 134217728
net.ipv4.tcp_wmem = 4096 1048576 134217728
net.ipv4.tcp_congestion_control = bbr
net.core.default_qdisc = fq
net.ipv4.tcp_mtu_probing = 1          # MTU discovery để tránh fragmentation
net.core.netdev_budget = 600          # packets xử lý mỗi CPU cycle
net.ipv4.tcp_max_syn_backlog = 8192
net.core.somaxconn = 65535
```

---

13. Security

B Network segmentation cơ bản

```
# Nguyên tắc: least privilege cho network
# - Web servers: chỉ nhận 80/443 từ internet
# - App servers: chỉ nhận từ web servers (không expose ra internet)
# - DB servers: chỉ nhận từ app servers, không có public IP

# Kiểm tra exposed ports
nmap -sS -O -p- 0.0.0.0/0   # ĐỪNG chạy lên internet không có phép
ss -tulpn | grep LISTEN       # xem từ bên trong server
```

I Zero Trust basics

```
Traditional (perimeter security):
  Internet → Firewall → "Trusted" Internal Network → Servers
  Vấn đề: ai vào được internal network là tin tưởng hoàn toàn

Zero Trust principles:
  1. Không tin tưởng ngầm định — verify mọi request
  2. Least privilege access — chỉ cấp đúng quyền cần thiết
  3. Assume breach — luôn giả định đã bị compromise

Implementation tools:
  - mTLS giữa services (Istio, Linkerd)
  - Service mesh với policy enforcement
  - Identity-aware proxy (BeyondCorp, Cloudflare Access)
```

Tailscale — mesh VPN zero config

```
# Cài và join network
curl -fsSL https://tailscale.com/install.sh | sh
tailscale up

# Xem devices trong network
tailscale status

# Exit node — route traffic qua device khác (như full-tunnel VPN)
tailscale up --exit-node=100.x.x.x

# Subnet routing — expose LAN subnet ra Tailscale network
tailscale up --advertise-routes=192.168.1.0/24
# Trên admin console: approve subnet route
# Devices khác: tailscale up --accept-routes

# ACL (access control) — giới hạn ai communicate với ai
# Quản lý qua Tailscale admin console với JSON policy
```

Nebula — self-hosted mesh VPN

```
# /etc/nebula/config.yaml — node config
pki:
  ca: /etc/nebula/ca.crt
  cert: /etc/nebula/node.crt
  key: /etc/nebula/node.key

static_host_map:
  "10.88.0.1": ["lighthouse.myapp.com:4242"]   # lighthouse IP → public address

lighthouse:
  am_lighthouse: false
  hosts:
    - "10.88.0.1"

listen:
  host: 0.0.0.0
  port: 4242

firewall:
  outbound:
    - port: any
      proto: any
      host: any
  inbound:
    - port: 22
      proto: tcp
      group: ops       # chỉ nodes thuộc group "ops" mới SSH được
```

Quick Reference

Tool	Dùng khi nào
`ss -tulpn`	Xem listening ports + process
`dig +trace domain`	Debug DNS resolution từng bước
`mtr --tcp --port 443`	Traceroute qua firewall chặn ICMP
`openssl s_client -connect`	Debug TLS/cert issues
`tcpdump -i any port 53`	Capture DNS traffic
`curl -w time_total`	Đo latency từng giai đoạn HTTP

Tool	Dùng khi nào
`conntrack -L`	Xem connection tracking table
`ip route get 8.8.8.8`	Xem route sẽ dùng cho destination
`nc -zv host port`	Test port connectivity nhanh
`ssh -J bastion target`	Nhảy qua bastion host

PART II

Server Optimization

Cách server nhỏ chịu tải 1M+ user đồng thời.

15

Server Optimization Architecture — Toàn cảnh tối ưu từ bit đến cluster

> **Tags:** B Beginner — I Intermediate — A Advanced
> **Series:** optimization-00 / 08 — INDEX FILE
> **Mục đích:** Big picture + roadmap. Đọc file này trước, rồi jump vào file phù hợp với scale hiện tại.

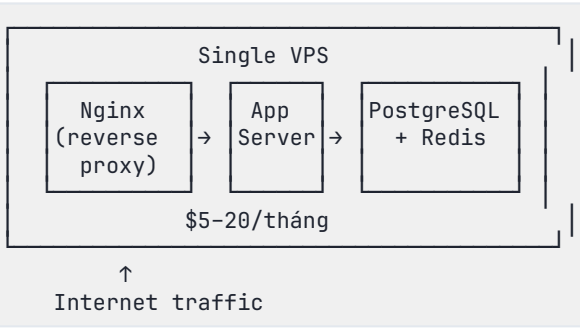
Triết lý tối ưu

Tối ưu sai thứ tự = lãng phí công sức. Thứ tự đúng:

```
1. Đo trước, đoán sau        → Profile trước khi optimize
2. Bottleneck cao nhất trước → 1 đêm làm index = tốt hơn 1 tuần rewrite
3. Tối ưu chi phí/hiệu quả   → 10 phút gzip > 2 tuần viết cache layer
4. Đừng over-engineer        → Level 1 setup không cần K8s
```

The Scale Journey

Level 1: Single VPS (1–10K users) B

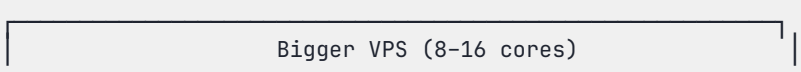


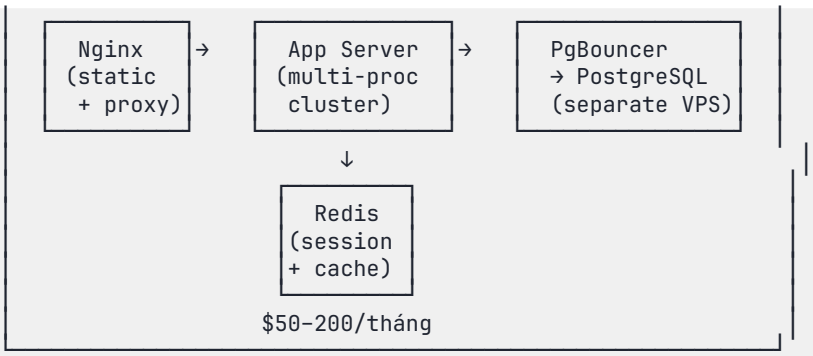
Bottleneck: slow queries, không cache, transfer thừa bandwidth.

Tối ưu cần làm: - Database indexes → optimization-05 - Basic caching (Redis) → optimization-04 - Gzip compression → optimization-06 - Nginx cơ bản (reverse proxy, static files) → optimization-08

Bỏ qua: kernel tuning, connection pooling, horizontal scaling — quá sớm, không cần.

Level 2: Vertical Scaling (10K–100K users) I

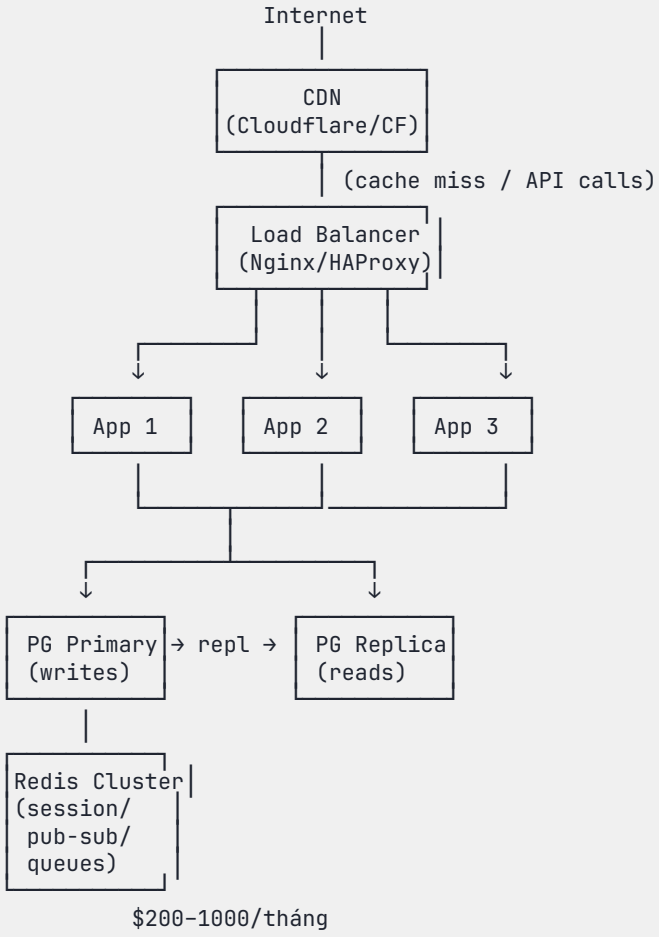




Bottleneck: kernel chặn connections, quá nhiều DB connections, in-process memory.

Tối ưu cần làm: - Kernel tuning (file descriptors, TCP stack) → optimization-02 - Connection pooling với PgBouncer → optimization-03 - In-process caching + cache stampede prevention → optimization-04 - Async processing cho heavy tasks → optimization-07

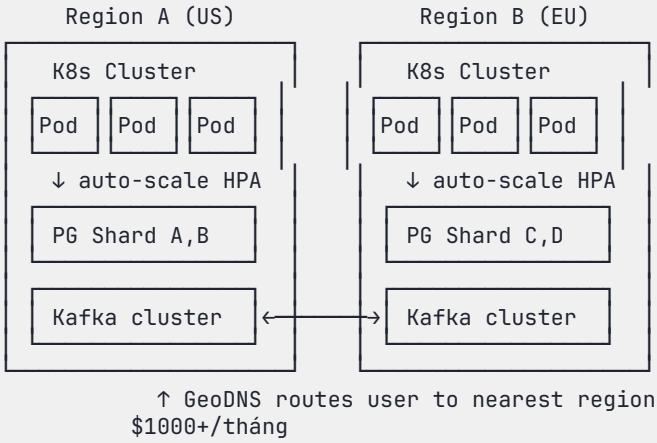
Level 3: Horizontal Scaling (100K–1M users) I A



Bottleneck: single-server limits, read-heavy DB, traffic spikes.

Tối ưu cần làm: - Load balancing → optimization-08 - Read replicas → optimization-05 - Async queuing (Redis Streams / RabbitMQ) → optimization-07 - CDN cho static assets → optimization-08 - Memory-efficient data structures (bitmap, HyperLogLog) → optimization-01

Level 4: Distributed System (1M+ users) A



Đặc điểm: DB sharding, microservices, event-driven, multi-region CDN, Kafka cho event streaming.

Đừng nhảy thẳng lên Level 4 từ Level 1. Mỗi level có chi phí kỹ thuật và vận hành riêng. Level 3 đủ cho hầu hết startup Series B trở xuống.

The Optimization Checklist (Priority Order)

Thứ tự dưới đây là theo **impact/effort ratio** — làm từ trên xuống:

#	Optimization	Effort	Impact	File
1	**Database indexes**	1h	100x query speedup	optimization-05
2	**Add caching layer**	2h	10–100x ít DB queries hơn	optimization-04
3	**Connection pooling**	1h	10x concurrent users	optimization-03
4	**Gzip compression**	10 phút	70% giảm bandwidth	optimization-06
5	**CDN cho static assets**	30 phút	80% ít request tới origin	optimization-08
6	**Kernel tuning**	1h	10x max concurrent connections	optimization-02
7	**Async processing**	4h	hấp thụ traffic spike 10x	optimization-07
8	**Memory-efficient structures**	2h	1000x ít RAM hơn cho tracking	optimization-01
9	**Read replicas**	4h	2–3x read capacity	optimization-05
10	**Horizontal scaling**	8h	capacity tăng tuyến tính	optimization-08

Quy tắc: Nếu chưa làm #1 (indexes), đừng nhảy xuống #6 (kernel tuning). Bottleneck sẽ luôn là database trước khi là kernel.

Quick Reference: Memory Math

Bảng so sánh naive vs optimized cho các bài toán phổ biến:

Bài toán	Naive	Optimized	Tiết kiệm
Track 1M user online status	1M Redis keys ≈ 64MB	Bitmap = 122KB	**500x**
Đếm unique visitors	SET user IDs ≈ 50MB	HyperLogLog = 12KB	**4000x**

Bài toán	Naive	Optimized	Tiết kiệm
1M TCP connections	Kernel mặc định = không thể	Kernel đã tune = ~3.5GB RAM	**Possible**
1000 DB connections	PG direct = 10GB RAM	PgBouncer pool 50 = 500MB	**20x**
10M API responses/ngày	Không cache = 10M DB queries	99% hit rate = 100K queries	**100x**
1M user sessions	Per-user Redis hash ≈ 500MB	Packed fields ≈ 50MB	**10x**
Duplicate detection 10M items	HashSet ≈ 500MB	Bloom filter = 1.2MB	**400x**

Quick Reference: Throughput Math

Bottleneck	Untuned	Tuned	Cách đạt
Nginx connections	1,024	1M+	worker_connections + kernel tuning
PostgreSQL queries/sec	1,000	50K+	indexes + pooling + caching
Redis ops/sec	100K	1M+	pipelining + cluster
Node.js req/sec	5K	50K+	cluster mode + in-process cache
API bandwidth/ngày	100GB	30GB	gzip + CDN + sparse fieldsets

Files trong Series

#	File	Nội dung	Tag
00	`optimization-00-architecture-overview.md`	File này — big picture và index	B
01	`optimization-01-memory-efficient-data-structures.md`	Bitmap, HyperLogLog, Bloom filter, packed arrays	I A
02	`optimization-02-kernel-os-tuning.md`	sysctl, file descriptors, TCP stack, epoll, BBR	I A
03	`optimization-03-connection-pooling.md`	PgBouncer, ProxySQL, app pools, HTTP keepalive	I
04	`optimization-04-caching-strategies.md`	L1/L2/L3 cache, patterns, invalidation, stampede	I A
05	`optimization-05-database-at-scale.md`	Indexes, partitioning, VACUUM, SKIP LOCKED, replicas	I A
06	`optimization-06-application-patterns.md`	Event loop, batching, compression, rate limiting	B I
07	`optimization-07-async-queuing.md`	Redis Streams, RabbitMQ, Kafka, event-driven	I A
08	`optimization-08-load-balancing-proxy.md`	Nginx/HAProxy tuning, SSL, DNS LB, horizontal scaling	I A

Cách đọc series này

```
Mới bắt đầu (Level 1)?
  → Đọc: 05 (DB) → 04 (cache) → 06 (app patterns)

Đang scale lên Level 2?
  → Đọc: 02 (kernel) → 03 (pooling) → 07 (async)

Chuẩn bị horizontal scaling?
  → Đọc: 08 (LB) → 01 (memory) → 05 phần replicas

Muốn hiểu toàn bộ?
  → Đọc từ 01 đến 08 theo thứ tự
```

Series này được viết cho VPS nhỏ ($5–200/tháng) đang phục vụ traffic thật. Không phải lý thuyết hàn lâm.

16

Memory-Efficient Data Structures — Xử lý hàng triệu entity với vài KB RAM

> **Triết lý cốt lõi:** Thay vì 1 key/row per entity (cách naive), dùng data structures nén thông tin xuống bit-level hoặc probabilistic level. Server nhỏ vẫn handle được 1M+ concurrent users nếu bạn không lãng phí RAM vào những thứ không cần thiết.

Tags: Production-tested patterns | Redis + PostgreSQL | Scale: 1M+ users
Độ khó: B Beginner · I Intermediate · A Advanced

Mục lục

1. Redis Bitmap
2. Redis HyperLogLog
3. Bloom Filter
4. Count-Min Sketch
5. Roaring Bitmaps
6. Packed Arrays & Bitfields
7. Trie / Radix Tree
8. Compact Encoding trong PostgreSQL
9. Interning & Deduplication Patterns
10. Cheat Sheet tổng hợp

1. Redis Bitmap

Tag: B

Khái niệm

Bitmap là một chuỗi bits. Redis lưu bitmap dưới dạng string, mỗi byte = 8 bits. Mỗi bit đại diện cho 1 entity (ví dụ: user ID). Đây là cách nén thông tin Boolean cực kỳ hiệu quả.

Tính toán bộ nhớ

```
1,000,000 users × 1 bit/user = 1,000,000 bits
= 125,000 bytes = ~122 KB

So sánh với SET (key per user):
1,000,000 keys × ~50 bytes/key (overhead) = ~50 MB
→ Bitmap tiết kiệm hơn 400 lần
```

Use case 1: Tracking user online/offline

```
# User 12345 vừa login → set bit
SETBIT online:users 12345 1

# User 12345 logout → clear bit
SETBIT online:users 12345 0

# Kiểm tra user 12345 có online không?
GETBIT online:users 12345
# → 1 (online) hoặc 0 (offline), O(1)

# Đếm tổng số user đang online
BITCOUNT online:users
# O(n) trên string, nhưng n chỉ là 122KB — cực nhanh
```

Before vs After: - Before: `SET user:12345:online 1` với TTL → 1M keys = 50MB+ RAM + key expiry overhead - After: 1 bitmap key `online:users` = 122KB — giảm 400x

Use case 2: Daily Active Users (DAU)

```
# Mỗi khi user active trong ngày → set bit theo uid
SETBIT dau:2026-05-27 12345 1
SETBIT dau:2026-05-27 67890 1

# Đếm DAU ngày hôm nay
BITCOUNT dau:2026-05-27
# → 2 (hoặc bất kỳ số nào)

# Đếm DAU từ bit 0 đến bit 999999 (đề phòng có uid > 1M)
BITCOUNT dau:2026-05-27 0 124999
# Lưu ý: BITCOUNT đếm theo byte range, không phải bit range
# byte 0-124999 = bit 0-999999
```

Use case 3: Retention Analysis

```
# User active cả 2 ngày → retained users
BITOP AND retained:2026-05-26_27 dau:2026-05-26 dau:2026-05-27

# Đếm số user retain
BITCOUNT retained:2026-05-26_27
# → retention count

# Công thức retention D1: retained / dau:day1
# Ví dụ: 80,000 retained / 200,000 DAU day1 = 40% D1 retention
```

BITOP hỗ trợ: AND, OR, XOR, NOT — đủ để làm cohort analysis phức tạp.

Use case 4: Feature Flags per User

```
# 1 bitmap per feature flag
SETBIT feature:dark_mode 12345 1       # user 12345 có dark mode
SETBIT feature:dark_mode 67890 0       # user 67890 không có

SETBIT feature:new_dashboard 12345 1   # user 12345 vào beta

# Check: user 12345 có feature dark_mode không?
GETBIT feature:dark_mode 12345
# O(1), cực nhanh — dùng trong mỗi API request hoàn toàn ổn
```

Production scenario: Hệ thống A/B test cho 5M users, 20 features. - Naive: `user:feature` hash → 100M entries - Bitmap: 20 bitmaps × 625KB = 12.5MB total

Use case 5: Streak Tracking + Login Calendar

```
# Dùng BITFIELD để track 365 ngày, 1 bit/ngày, per user
# Mỗi user có 1 key, 365 bits = 46 bytes

# User 12345 login ngày thứ 147 trong năm
SETBIT streak:2026:12345 147 1

# Đọc 7 ngày gần nhất (từ bit 140 đến 146)
GETDEL streak:2026:12345  # (cần custom logic, dùng LUA script thực tế)

# Đếm tổng số ngày login trong năm
BITCOUNT streak:2026:12345
# → số ngày active trong năm

# Tính streak hiện tại: scan ngược từ ngày hiện tại
# → cần Lua script hoặc xử lý ở app layer
```

Memory cho 1M users × 365 ngày:

```
1M users × 46 bytes/user = 46MB
So sánh: 1M rows × 365 records trong SQL = 365M rows = hàng chục GB
→ Bitmap tiết kiệm 1000x+
```

❗ Gotchas — Bitmap

```
❗ Sparse bitmap: nếu user ID không sequential (e.g., UUID-based hoặc uid bắt đầu
    từ 10,000,000), bitmap sẽ allocate toàn bộ space từ bit 0 → bit max_uid.
    uid=10,000,000 → bitmap = 1.25MB chỉ cho 1 user.

    Giải pháp:
    1. ID mapping layer: map uid → sequential index trước khi SETBIT
       HSET uid:map 10000000 42  # uid 10M → compact index 42
    2. Dùng Roaring Bitmap (xem mục 5)
    3. Sharding: bitmap:shard:0 cho uid 0-999999, bitmap:shard:1 cho uid 1M-2M, v.v.

❗ BITCOUNT trả về số bit được set, không phải vị trí. Để lấy list uid active,
    cần scan toàn bộ bitmap và check từng bit — không scalable với 1M+ users.
    → HyperLogLog nếu chỉ cần count, không cần list.

❗ Redis string max size = 512MB = 4 tỷ bits. Đủ cho uid lên đến 4 tỷ.
```

---

2. Redis HyperLogLog

Tag: 🔵

Khái niệm

HyperLogLog (HLL) là probabilistic data structure để đếm **cardinality** (số phần tử unique) với sai số ~0.81% và bộ nhớ cố định **12KB bất kể có bao nhiêu phần tử**.

Tính toán bộ nhớ & sai số

```
Redis HLL implementation:
- 2^14 = 16,384 registers
- Mỗi register: 6 bits
- Tổng: 16,384 × 6 bits = 98,304 bits = 12,288 bytes ≈ 12 KB

Standard Error = 1.04 / sqrt(2^14) = 1.04 / 128 = 0.00813 ≈ 0.81%

Ví dụ thực tế:
- 1M unique visitors, HLL count: 991,900 – 1,008,100 (±0.81%)
```

```
- Sai số này hoàn toàn chấp nhận được cho analytics

Memory comparison:
SET với 1M UIDs: 1M × 8 bytes (int64) = 8MB (chưa kể Redis overhead ~50 bytes/member)
Thực tế SET: 1M members × ~50 bytes = ~50MB
HyperLogLog: 12KB
→ Tiết kiệm 4,000x
```

Commands cơ bản

```
# Thêm elements vào HLL
PFADD pageviews:2026-05-27 "user:12345"
PFADD pageviews:2026-05-27 "user:67890"
PFADD pageviews:2026-05-27 "user:12345"  # duplicate — ignored

# Đếm unique visitors hôm nay
PFCOUNT pageviews:2026-05-27
# → 2 (không phải 3, vì 12345 xuất hiện 2 lần)

# Thêm bằng nhiều elements cùng lúc
PFADD api:callers:2026-05-27 "ip:1.2.3.4" "ip:5.6.7.8" "ip:1.2.3.4"
PFCOUNT api:callers:2026-05-27
# → 2
```

Use case: Unique visitors theo giờ → ngày → tuần

```
# Track theo giờ
PFADD uv:2026-05-27:14 "user:12345"  # 14h ngày 27/5
PFADD uv:2026-05-27:15 "user:67890"  # 15h ngày 27/5

# Merge hourly → daily unique (không cộng đơn giản — dùng PFMERGE)
PFMERGE uv:2026-05-27 uv:2026-05-27:00 uv:2026-05-27:01 ... uv:2026-05-27:23
PFCOUNT uv:2026-05-27
# → unique visitors trong cả ngày (dedup across hours)

# Merge daily → weekly
PFMERGE uv:2026-W22 uv:2026-05-25 uv:2026-05-26 uv:2026-05-27
PFCOUNT uv:2026-W22
# → unique visitors trong tuần
```

Tại sao không cộng đơn giản? Nếu user A visit cả ngày thứ 2 lẫn thứ 3, cộng DAU thô sẽ đếm họ 2 lần. PFMERGE xử lý dedup đúng cách.

Use case: Unique IPs per API endpoint

```
# Rate limiting monitor: unique IPs calling /api/payments
PFADD api:unique_ips:/api/payments $(date +%Y-%m-%d-%H) "ip:1.2.3.4"

# Alert nếu unique IPs tăng đột biến (DDoS detection)
PFCOUNT api:unique_ips:/api/payments:2026-05-27-14
# So sánh với giờ trước, nếu tăng 10x → alert
```

Gotchas — HyperLogLog

```
Không thể lấy ra danh sách các phần tử đã thêm vào — chỉ đếm được.
   Nếu cần "ai là unique visitors?", dùng Set hoặc Bloom Filter.

Không thể xóa element khỏi HLL. Nếu cần delete, phải rebuild từ đầu.
   → Dùng time-windowed keys (hourly/daily) thay vì 1 key duy nhất.

Sai số 0.81% có thể bị amplify ở low cardinality.
   HLL báo 100 unique users thực ra có thể là 99 hoặc 101 — sai số lớn tương đối.
   → Với cardinality < 1000, dùng Set thông thường.
```

```
PFMERGE tạo union, không có intersection. Không tính được
   "users active cả 2 ngày" bằng HLL — dùng Bitmap BITOP AND cho đó.
```

---

3. Bloom Filter

Tag: I

Khái niệm

Bloom Filter trả lời câu hỏi: "Phần tử này **có thể** đã tồn tại, hay **chắc chắn** chưa tồn tại?" — với bộ nhớ nhỏ hơn Set hàng chục lần.

- **False positive:** Nói "có" nhưng thực ra chưa có (xảy ra với xác suất p)
- **False negative:** Không bao giờ xảy ra — nếu nói "chưa có" thì chắc chắn chưa có

Tính toán bộ nhớ

```
Công thức tối ưu:
m = -n × ln(p) / (ln(2))²

Trong đó:
  n = số phần tử dự kiến
  p = tỷ lệ false positive mong muốn
  m = số bits cần thiết

Ví dụ: 1M elements, false positive 1%:
m = -1,000,000 × ln(0.01) / (ln(2))²
m = -1,000,000 × (-4.605) / 0.480
m = 9,585,058 bits ≈ 1.15 MB

Số hash functions tối ưu:
k = (m/n) × ln(2) = (9.585) × 0.693 ≈ 6.6 → dùng 7

So sánh:
Set với 1M strings (~20 chars): 1M × 70 bytes = ~70 MB
Bloom Filter 1% FP: 1.15 MB
→ Tiết kiệm 60x
```

Dùng Redis Bloom (module RedisBloom)

```
# Tạo Bloom Filter
BF.RESERVE notifications:seen 0.01 1000000
# error_rate=0.01 (1%), capacity=1M

# Thêm element
BF.ADD notifications:seen "user:12345:notif:789"
# → 1 (mới), 0 (đã có)

# Kiểm tra
BF.EXISTS notifications:seen "user:12345:notif:789"
# → 1 (có thể đã thấy) hoặc 0 (chắc chắn chưa thấy)

# Bulk add
BF.MADD notifications:seen "user:1:notif:1" "user:2:notif:2" "user:3:notif:3"

# Thông tin bộ nhớ
BF.INFO notifications:seen
```

Use case 1: Notification dedup — "User đã thấy notification này chưa?"

```
# Trước khi gửi push notification
def should_send_notification(user_id: int, notif_id: int) → bool:
    key = f"user:{user_id}:notif:{notif_id}"

    # Bloom Filter check — O(1), sub-millisecond
    if redis.execute_command("BF.EXISTS", "notifications:seen", key):
        return False  # Có thể đã thấy — skip (1% false skip là chấp nhận được)

    # Chắc chắn chưa thấy — gửi đi
    redis.execute_command("BF.ADD", "notifications:seen", key)
    return True

# Production: 10M notifications/day, 1% false positive = 100K bị skip oan
# Trade-off: 100K skipped vs không cần 700MB Set — acceptable với non-critical notifications
```

Use case 2: URL dedup trong web crawler

```
# Crawler: đã crawl URL này chưa?
def should_crawl(url: str) → bool:
    if redis.execute_command("BF.EXISTS", "crawler:visited", url):
        return False  # Skip — đã crawl (hoặc false positive, bỏ qua cũng được)
    redis.execute_command("BF.ADD", "crawler:visited", url)
    return True

# 100M URLs, 1% FP → 12MB Bloom Filter vs 7GB URL Set
```

Use case 3: Username availability check (trước khi query DB)

```
def is_username_available(username: str) → bool:
    # Check Bloom Filter trước — nếu "chắc chắn không tồn tại" → available ngay
    if not redis.execute_command("BF.EXISTS", "usernames:taken", username):
        return True  # 100% chưa tồn tại — không cần query DB

    # Có thể tồn tại → query DB để xác nhận (tránh false positive)
    return not db.query("SELECT 1 FROM users WHERE username = %s", username)
```

Pattern này giảm DB query 60-80% — chỉ query DB khi Bloom Filter nói "có thể tồn tại".

Cuckoo Filter vs Bloom Filter

```
Bloom Filter:
- Không hỗ trợ deletion
- Bộ nhớ nhỏ hơn một chút
- Lookup nhanh hơn một chút

Cuckoo Filter:
- Hỗ trợ deletion (CF.DEL)
- Slightly more memory (~15-20%)
- Dùng khi cần remove elements (e.g., user revoke session)

Redis commands:
CF.RESERVE, CF.ADD, CF.EXISTS, CF.DEL, CF.COUNT
```

⚠ Gotchas — Bloom Filter

```
⚠ Không thể retrieve elements đã add vào — chỉ check membership.

⚠ False positive rate tăng khi số elements vượt quá capacity.
    Khi BF.INFO báo "Number of items inserted" gần "Capacity" → tạo BF mới
    hoặc scale up capacity.

⚠ Không thể shrink BF. Phải set capacity đủ lớn ngay từ đầu.
```

```
    Rule of thumb: capacity × 2 so với dự kiến.

! Nhiều hash functions → độ chính xác cao hơn nhưng write chậm hơn.
    k=7 là sweet spot cho production.

! Bloom Filter không replace database — chỉ là guard layer.
    False negatives không xảy ra, nhưng false positives cần được handled.
```

---

4. Count-Min Sketch

Tag: A

Khái niệm

Count-Min Sketch (CMS) ước lượng **tần suất** của từng phần tử trong data stream, dùng 2D array nhỏ thay vì lưu counter cho từng phần tử riêng.

Tính toán bộ nhớ

```
CMS là một ma trận width × depth, mỗi ô là 4-byte counter.

Công thức:
  width  = ceil(e / epsilon)     # e ≈ 2.718
  depth  = ceil(ln(1 / delta))   # delta = xác suất sai

Ví dụ thực tế — rate limiting cho 10M users, sai số 0.1%, confidence 99%:
  epsilon = 0.001 (sai số tương đối)
  delta   = 0.01  (1% xác suất vượt sai số)
  width   = ceil(2.718 / 0.001) = 2718
  depth   = ceil(ln(100)) = ceil(4.605) = 5

  Memory = 2718 × 5 × 4 bytes = 54,360 bytes ≈ 53 KB

So sánh với HashMap counter per user:
  10M users × 8 bytes (int64 counter) = 80 MB (chưa tính HashMap overhead)
  Thực tế HashMap: 10M entries × ~50 bytes = ~500 MB
  CMS: 53 KB
  → Tiết kiệm 10,000x
```

Commands (Redis CMS module)

```
# Tạo CMS với epsilon=0.001, delta=0.01
CMS.INITBYPROB rate_limit:actions 0.001 0.01

# Thêm events (increment counter)
CMS.INCRBY rate_limit:actions "user:12345:api_call" 1
CMS.INCRBY rate_limit:actions "user:67890:api_call" 1
CMS.INCRBY rate_limit:actions "user:12345:api_call" 1  # lần 2

# Query tần suất
CMS.QUERY rate_limit:actions "user:12345:api_call"
# → [2] (có thể over-estimate nhưng không under-estimate)

CMS.QUERY rate_limit:actions "user:12345:api_call" "user:67890:api_call"
# → [2, 1]

# Merge nhiều CMS (e.g., từ nhiều shard)
CMS.MERGE combined 2 shard:1 shard:2
```

Use case 1: Rate Limiting không cần counter per user

```python
import redis

r = redis.Redis()

def check_rate_limit(user_id: int, limit: int = 100) -> bool:
    """
    Rate limit: 100 actions/minute per user.
    CMS over-estimates → conservative rate limiting (users có thể bị limit sớm hơn 0.1%)
    """
    key = f"user:{user_id}:minute:{int(time.time() / 60)}"

    # Increment và query trong 1 bước
    r.execute_command("CMS.INCRBY", "rate_limit", key, 1)
    count = r.execute_command("CMS.QUERY", "rate_limit", key)[0]

    return count ⩽ limit

# 10M users, 1 CMS = 53KB vs 10M Redis keys = 500MB+
```

Use case 2: Trending Topics Detection

```python
def track_event(topic: str):
    """Track trending topics trong sliding window."""
    window = int(time.time() / 300)  # 5-minute windows
    r.execute_command("CMS.INCRBY", f"trending:{window}", topic, 1)

def get_trending():
    """Lấy top topics — kết hợp CMS với Top-K."""
    # CMS cho frequency estimate
    # Top-K structure (Redis TOPK module) cho heavy hitters
    pass

# Production: Twitter-scale trending, 100K topics/minute
# CMS + TOPK: vài MB vs exact counter: hàng GB
```

Use case 3: Heavy Hitter Detection (DDoS / Abuse)

```python
def detect_heavy_hitter(ip: str, threshold: int = 1000) -> bool:
    """
    Phát hiện IP gửi > 1000 requests/phút (potential DDoS).
    CMS sẽ over-estimate → false positive (block oan) có thể xảy ra 0.1%.
    Acceptable: temporary block, auto-unblock sau 1 phút.
    """
    window = int(time.time() / 60)
    r.execute_command("CMS.INCRBY", f"req_count:{window}", ip, 1)
    count = r.execute_command("CMS.QUERY", f"req_count:{window}", ip)[0]
    return count > threshold
```

❗ Gotchas — Count-Min Sketch

```
❗ CMS chỉ over-estimate, không bao giờ under-estimate.
   Điều này có nghĩa: rate limiter dùng CMS sẽ block users sớm hơn thực tế một chút.
   Acceptable nếu sai số nhỏ (0.1%). Không dùng cho billing/invoicing.

❗ Không thể remove/decrement counter trong standard CMS.
   Conservative CMS (CCMS) hỗ trợ decrement nhưng ít phổ biến hơn.

❗ Collision giữa các keys làm tăng false positive rate.
   Tăng width để giảm collision, tăng depth để giảm xác suất vượt sai số.

❗ CMS không cho biết "ai là top K" — chỉ estimate frequency khi query.
   Kết hợp với Redis TOPK module để tìm heavy hitters tự động:
   TOPK.ADD, TOPK.LIST, TOPK.QUERY
```

5. Roaring Bitmaps

Tag: A

Khái niệm

Regular bitmap với 10M users tốn 1.25MB dù chỉ có 1000 users thực sự active. Roaring Bitmap giải quyết vấn đề sparse bitmap bằng cách dùng compressed containers.

Tính toán bộ nhớ

```
Regular bitmap: 10,000,000 UIDs → 10M bits / 8 = 1,250,000 bytes = 1.25 MB
(Dù chỉ có 1000 users active)

Roaring Bitmap với 1000 users rải rác trong khoảng 0-10M:
- Chia thành chunks 65,536 (2^16) values mỗi chunk
- Mỗi chunk dùng container phù hợp:
  * Array container: <4096 elements → 2 bytes/element
  * Bitmap container: >4096 elements → 8KB fixed
  * Run container (RLE): consecutive runs → 4 bytes/run

1000 users rải rác → Array containers:
~15 chunks × 70 users/chunk × 2 bytes = ~2,100 bytes ≈ 2 KB
→ Tiết kiệm 600x so với regular bitmap
```

Libraries

```
# Python
pip install pyroaring

# Java/Kotlin
# Maven: org.roaringbitmap:RoaringBitmap

# Rust
# Cargo: roaring = "0.10"

# C
# CRoaring: https://github.com/RoaringBitmap/CRoaring

# Go
# github.com/RoaringBitmap/roaring
```

Python example

```python
from pyroaring import BitMap

# Tạo bitmap
active_users = BitMap()

# Thêm UIDs (không cần sequential)
active_users.add(1)
active_users.add(1_000_000)   # Sparse — không tốn 1MB gap
active_users.add(9_999_999)

print(len(active_users))       # → 3
print(1_000_000 in active_users)  # → True

# Set operations — cực nhanh (SIMD-optimized)
dau_yesterday = BitMap([1, 2, 3, 1_000_000])
dau_today     = BitMap([2, 3, 1_000_000, 5_000_000])

retained = dau_yesterday & dau_today   # Intersection
churned  = dau_yesterday - dau_today   # Difference
```

```
all_seen = dau_yesterday | dau_today   # Union

print(len(retained))   # → 3 (user 2, 3, 1M)
print(len(churned))    # → 1 (user 1)

# Serialize để lưu vào Redis hoặc disk
serialized = active_users.serialize()
redis.set("active:roaring", serialized)

# Deserialize
loaded = BitMap.deserialize(redis.get("active:roaring"))
```

Use case: Search engine posting lists

```
# Inverted index: từ "python" xuất hiện trong các document nào?
from pyroaring import BitMap

posting_lists = {
    "python": BitMap([1, 5, 23, 10_000, 9_999_999]),
    "redis":  BitMap([1, 23, 50_000]),
    "bitmap": BitMap([5, 23, 9_999_999]),
}

# Tìm docs chứa cả "python" AND "redis"
result = posting_lists["python"] & posting_lists["redis"]
# → BitMap([1, 23]) — 2 docs

# Tìm docs chứa "python" OR "redis"
result = posting_lists["python"] | posting_lists["redis"]
# → BitMap([1, 5, 23, 10_000, 50_000, 9_999_999])

# Memory: 1B documents, average 10K docs/term
# Regular bitmap: 1B bits / 8 = 125 MB per term
# Roaring: ~20 KB per term (10K docs × 2 bytes, array container)
```

Use case: Permission bitmasks

```
# RBAC phức tạp: user thuộc nhiều groups, mỗi group có set permissions
from pyroaring import BitMap

# User IDs trong mỗi group
group_admin    = BitMap([1, 2, 3, 100])
group_editor   = BitMap([4, 5, 6, 100, 200])
group_viewer   = BitMap(range(1000, 10000))  # 9000 users

# User 100 thuộc group nào?
user_id = 100
groups_of_user = {
    name: uid_set
    for name, uid_set in {"admin": group_admin, "editor": group_editor, "viewer": group_viewer}.i
tems()
    if user_id in uid_set
}
# → {'admin': ..., 'editor': ...}

# Tìm tất cả users có quyền admin OR editor
privileged = group_admin | group_editor
print(len(privileged))  # → 8 unique users
```

Gotchas — Roaring Bitmaps

```
Roaring Bitmap không có native Redis module (không như Bloom/CMS).
    Phải serialize/deserialize khi lưu vào Redis:
    redis.set(key, bitmap.serialize())
    bitmap = BitMap.deserialize(redis.get(key))
    → Không atomic, cần WATCH/MULTI/EXEC cho concurrent writes.
```

```
⚠ Serialize/deserialize overhead với large bitmaps có thể thành bottleneck.
   Cache deserialized bitmap ở application layer, không deserialize mỗi request.

⚠ Roaring Bitmap tốt nhất cho sparse data. Dense data (80%+ bits set) thì
   regular bitmap hoặc bitmap container trong Roaring đều tốt như nhau.

⚠ Thread safety: hầu hết implementations không thread-safe.
   Dùng locks hoặc immutable snapshots cho concurrent access.
```

---

6. Packed Arrays & Bitfields

Tag: I

Khái niệm

Thay vì mỗi user có 1 Redis key riêng cho mỗi counter nhỏ, pack nhiều counters vào 1 string dùng BITFIELD — mỗi counter chỉ chiếm đúng số bits cần thiết.

Tính toán bộ nhớ

```
Ví dụ: rate limit counter per user, max 255 requests/minute
  → cần 8 bits (u8) per user

1M users × 1 separate Redis key:
  1M keys × ~50 bytes overhead = 50 MB

1M users packed với BITFIELD:
  1M × 8 bits = 1M bytes = 1 MB + 1 key overhead
  → Tiết kiệm 50x

Ví dụ khác: user level 0-15 (4 bits, u4)
  1M users × 4 bits = 500 KB
  vs 1M separate keys = 50 MB
  → Tiết kiệm 100x
```

Redis BITFIELD commands

```
# Cấu trúc: BITFIELD key [GET type offset] [SET type offset value] [INCRBY type offset increment]
# type: u8 (uint 8-bit), i8 (int 8-bit), u16, i32, v.v.
# offset: bit position, hoặc #<index> (nhân với type size)

# Set counter cho user 12345 (u8, tối đa 255)
BITFIELD rate_counters SET u8 #12345 0

# Increment counter user 12345
BITFIELD rate_counters INCRBY u8 #12345 1
# → [1]

# Increment lại 5 lần
BITFIELD rate_counters INCRBY u8 #12345 5
# → [6]

# Get counter user 12345
BITFIELD rate_counters GET u8 #12345
# → [6]

# Overflow policy: WRAP (default, wraps at 0), SAT (saturates at max), FAIL (returns nil)
BITFIELD rate_counters OVERFLOW SAT INCRBY u8 #12345 300
# → [255] (saturated, không wrap về 0)

# Atomic multi-operation
BITFIELD rate_counters \
  OVERFLOW SAT \
```

```
  INCRBY u8 #12345 1 \
  GET u8 #12345
# → [7, 7] — increment và get trong 1 round trip
```

Use case 1: Rate limit counters

```python
import redis
import time

r = redis.Redis()

def check_and_increment_rate_limit(user_id: int, limit: int = 100) -> bool:
    """
    Rate limit: 100 requests/minute per user.
    Dùng BITFIELD packed array — 1MB cho 1M users thay vì 50MB separate keys.
    """
    # Key per minute window
    minute_window = int(time.time() / 60)
    key = f"rl:{minute_window}"

    # Atomic: increment và get, saturate at 255 (u8 max)
    results = r.execute_command(
        "BITFIELD", key,
        "OVERFLOW", "SAT",
        "INCRBY", "u8", f"#{user_id}", 1,
        "GET", "u8", f"#{user_id}"
    )

    new_count = results[1]

    # TTL: expire sau 2 phút (1 window + buffer)
    r.expire(key, 120)

    return new_count ⩽ limit

# Note: u8 max = 255. Nếu limit > 255, dùng u16 (max 65535) → 2 bytes/user → 2MB/1M users
```

Use case 2: User level/tier tracking

```python
# User tiers: 0=free, 1=basic, 2=pro, 3=enterprise (cần 2 bits)
# 1M users × 2 bits = 250 KB

def get_user_tier(user_id: int) -> int:
    result = r.execute_command("BITFIELD", "user_tiers", "GET", "u2", f"#{user_id}")
    return result[0]

def set_user_tier(user_id: int, tier: int):
    r.execute_command("BITFIELD", "user_tiers", "SET", "u2", f"#{user_id}", tier)

# Upgrade user to pro
set_user_tier(12345, 2)
print(get_user_tier(12345))  # → 2
```

Use case 3: A/B test assignment tracking

```
# 1 bit per user per experiment = 1M users, 10 experiments = 10 × 122KB = 1.22MB
# Experiment 0: control vs treatment
BITFIELD ab:exp:0 SET u1 #12345 1   # user 12345 → treatment group
BITFIELD ab:exp:0 GET u1 #12345     # → [1]
```

Gotchas — BITFIELD

```
BITFIELD dùng bit offset #<index> là offset trong units của type size.
    BITFIELD key GET u8 #5 → bits từ position 40 (5 × 8).
    BITFIELD key GET u8 5  → bits từ position 5 (absolute bit offset).
```

```
    Nhầm 2 dạng này gây bug khó debug.

! u8 max = 255. Nếu counter vượt 255 với OVERFLOW WRAP → restart từ 0.
    Luôn dùng OVERFLOW SAT cho rate limiting.

! Concurrent INCRBY là atomic trong Redis (single-threaded), nhưng khi dùng
    Redis Cluster, cần đảm bảo cùng key về cùng shard (dùng hash tags: {rl}:12345).

! Xóa counter: SET về 0, không thể "free" individual bits trong string.
    Để giải phóng memory, dùng time-windowed keys với TTL (xem example trên).
```

---

7. Trie / Radix Tree

Tag: A

Khái niệm

Trie (prefix tree) lưu strings bằng cách chia sẻ prefix chung giữa các keys. Lookup là O(key_length), không phụ thuộc vào số lượng keys. Radix Tree (compact trie) merge các nodes chỉ có 1 child để tiết kiệm memory.

Tính toán bộ nhớ

```
Ví dụ: 1M URLs bắt đầu với "https://example.com/"
  Regular Set: 1M × 30 chars average = 30 MB
  Trie: prefix "https://example.com/" (20 chars) chia sẻ = 20 bytes × 1 node
        remaining paths: 1M × 10 chars average = 10 MB
        Node overhead: 1M nodes × ~40 bytes = 40 MB
        → Total: ~50 MB (không tốt hơn với URL đa dạng)

Ví dụ: IP routing table (ISP-scale)
  100K prefixes như "192.168.1.0/24"
  Trie: tối đa 32 levels (IPv4), mỗi node là 1 bit
        Dense tree = efficient traversal, O(32) = O(1) lookup
        ~100K × 32 nodes × 16 bytes = ~50 MB
        Radix Tree: merge single-child nodes → ~3-5 MB

Advantage chính của Trie: PREFIX MATCHING và O(key_length) lookup
```

Python Trie implementation

```python
from typing import Optional, Dict

class TrieNode:
    __slots__ = ('children', 'is_end', 'value')

    def __init__(self):
        self.children: Dict[str, 'TrieNode'] = {}
        self.is_end: bool = False
        self.value = None

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, key: str, value=None):
        node = self.root
        for char in key:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end = True
        node.value = value
```

```
    def search(self, key: str) -> Optional[object]:
        node = self.root
        for char in key:
            if char not in node.children:
                return None
            node = node.children[char]
        return node.value if node.is_end else None

    def starts_with(self, prefix: str) -> list:
        """Autocomplete: tìm tất cả keys có prefix cho trước."""
        node = self.root
        for char in prefix:
            if char not in node.children:
                return []
            node = node.children[char]

        # DFS để lấy tất cả keys
        results = []
        self._dfs(node, prefix, results)
        return results

    def _dfs(self, node: TrieNode, prefix: str, results: list):
        if node.is_end:
            results.append(prefix)
        for char, child in node.children.items():
            self._dfs(child, prefix + char, results)

# Ví dụ: autocomplete
trie = Trie()
for route in ["/api/users", "/api/users/profile", "/api/posts", "/api/payments"]:
    trie.insert(route)

print(trie.starts_with("/api/u"))
# → ['/api/users', '/api/users/profile']
```

Use case: IP geolocation lookup

```
# IP range lookup dùng radix tree — O(32) lookup thay vì binary search O(log n)
# Production: sử dụng library có sẵn

import ipaddress
from pytricia import PyTricia  # pip install pytricia

# Build routing table
trie = PyTricia()
trie["192.168.0.0/16"] = {"country": "VN", "city": "Ho Chi Minh"}
trie["10.0.0.0/8"]     = {"country": "private", "city": None}
trie["1.2.3.0/24"]     = {"country": "AU", "city": "Sydney"}

# Lookup — O(32) không phụ thuộc size of table
result = trie["192.168.1.100"]
print(result)  # → {"country": "VN", "city": "Ho Chi Minh"}

# 100K IP ranges → lookup 100ns vs HashMap O(1) nhưng cần exact match
# Trie cho longest-prefix matching — HashMap không làm được
```

Redis Streams dùng Radix Tree nội bộ

Redis Streams lưu message IDs trong radix tree internally, cho phép O(log n) insertion và range queries (XRANGE, XREAD) hiệu quả. Đây là lý do Streams scale tốt hơn List cho ordered data.

⚠ Gotchas — Trie

```
⚠ Trie tốt cho PREFIX operations. Nếu chỉ cần exact match, HashMap nhanh hơn.

⚠ Memory overhead cao khi keys đa dạng (ít prefix chung).
    Worst case: mỗi key là 1 branch duy nhất = overhead gấp đôi vs HashMap.
```

⚠ Python Trie thuần Python chậm. Dùng C extension (pytricia, marisa-trie) cho production.

⚠ Radix Tree phức tạp hơn Trie để implement nhưng memory efficient hơn nhiều.
Trong production, dùng battle-tested libraries thay vì tự implement.

---

8. Compact Encoding trong PostgreSQL

Tag: B

Chọn đúng kiểu dữ liệu

```
-- SMALLINT (2 bytes) vs INT (4 bytes) vs BIGINT (8 bytes)
-- 1M rows, column user_age:
-- SMALLINT: 2 MB | INT: 4 MB | BIGINT: 8 MB

CREATE TABLE users (
    id          BIGINT PRIMARY KEY,          -- Cần BIGINT cho PK (>2B users)
    age         SMALLINT NOT NULL,           -- 0-32767 đủ cho tuổi
    score       SMALLINT DEFAULT 0,          -- 0-1000 score → SMALLINT (không dùng INT)
    status      SMALLINT NOT NULL DEFAULT 0, -- 0=active,1=suspended,2=deleted
    tier        SMALLINT NOT NULL DEFAULT 0  -- 0=free,1=basic,2=pro,3=enterprise
);

-- Tính toán: 1M users
-- Dùng INT cho age,score,status,tier: 4 × 4 bytes × 1M = 16 MB extra
-- Dùng SMALLINT: 4 × 2 bytes × 1M = 8 MB
-- Tiết kiệm: 8 MB + index savings
```

ENUM type

```
-- VARCHAR cho status: variable length, 10-20 bytes + overhead
-- ENUM: 4 bytes fixed (internal OID)

CREATE TYPE user_status AS ENUM ('active', 'suspended', 'deleted', 'pending');

CREATE TABLE users (
    id     BIGINT PRIMARY KEY,
    status user_status NOT NULL DEFAULT 'active'
);

-- 1M rows: ENUM 4 bytes = 4MB vs VARCHAR avg 15 bytes = 15MB
-- Tiết kiệm 11MB + faster comparisons (int comparison vs string)

-- ⚠ Thêm ENUM value: ALTER TYPE ... ADD VALUE — không rollback được dễ dàng
-- ⚠ Rename ENUM value cần migration phức tạp hơn VARCHAR
```

Bit columns cho flags

```
-- Boolean trong PostgreSQL: 1 byte (không phải 1 bit!)
-- 10 boolean flags per user: 10 bytes
-- Dùng BIT(10): 10 bits = 2 bytes (hoặc integer bitmask)

-- Option 1: BIT column
CREATE TABLE user_features (
    user_id   BIGINT PRIMARY KEY,
    flags     BIT(16) NOT NULL DEFAULT B'0000000000000000'
    -- bit 0 = dark_mode, bit 1 = beta_access, bit 2 = notifications_email, ...
);

-- Set flag bit 1 (beta_access) cho user 12345
UPDATE user_features
SET flags = flags | B'0000000000000010'
```

```
WHERE user_id = 12345;

-- Check dark_mode (bit 0)
SELECT (flags & B'0000000000000001') = B'0000000000000000' AS has_dark_mode
FROM user_features WHERE user_id = 12345;

-- Option 2: INTEGER bitmask (thường dùng hơn vì dễ thao tác)
ALTER TABLE users ADD COLUMN feature_flags INTEGER NOT NULL DEFAULT 0;

-- Set dark_mode flag (bit 0)
UPDATE users SET feature_flags = feature_flags | 1 WHERE user_id = 12345;

-- Check dark_mode
SELECT (feature_flags & 1) > 0 AS has_dark_mode FROM users WHERE user_id = 12345;

-- 1M users, 10 boolean cols: 10 bytes/user = 10 MB
-- 1M users, 1 integer bitmask: 4 bytes/user = 4 MB
-- Tiết kiệm: 6 MB + simpler schema
```

JSONB overhead awareness

```
-- JSONB có overhead: 4 bytes per key-value pair + string storage
-- Với hot data (accessed frequently), tránh JSONB

-- Xấu: lưu user preferences trong JSONB
CREATE TABLE users (preferences JSONB);
-- {"theme":"dark","lang":"vi","notifications":true} → ~60 bytes

-- Tốt: separate columns nếu schema cố định
CREATE TABLE user_preferences (
    user_id         BIGINT PRIMARY KEY,
    theme           VARCHAR(20) DEFAULT 'light',
    language        CHAR(2) DEFAULT 'en',
    notifications   BOOLEAN DEFAULT true
);
-- → ~25 bytes, index-able, type-safe

-- JSONB hợp lý khi: schema thay đổi thường, semi-structured data,
-- ít truy vấn theo từng field cụ thể
```

Array columns thay junction table

```
-- Xấu: junction table cho tags (nếu chỉ cần simple membership)
CREATE TABLE post_tags (post_id BIGINT, tag_id INT);
-- 1M posts × 5 tags = 5M rows, 16 bytes/row = 80 MB + index overhead

-- Tốt: array column cho simple cases
CREATE TABLE posts (
    id   BIGINT PRIMARY KEY,
    tags INT[] DEFAULT '{}'
);
-- 1M posts × 5 tags × 4 bytes = 20 MB, GIN index cho @> operator

-- Query: tìm posts có tag_id = 42
CREATE INDEX idx_posts_tags ON posts USING GIN(tags);
SELECT * FROM posts WHERE tags @> ARRAY[42];

-- ⚠ Array không có foreign key constraint → app-level integrity
-- ⚠ Update 1 element trong array cần rewrite toàn row — không efficient với large arrays
-- Array columns hợp lý: ≤ 10-20 elements, ít update, không cần FK
```

⚠ Gotchas — PostgreSQL Compact Encoding

```
⚠ TOAST: PostgreSQL tự động nén và externalize values > 2KB (threshold configurable).
    TOAST giúp với large values nhưng adds read overhead.
    Với hot data (mỗi request đọc), avoid values thường xuyên TOASTed.
```

```
! CHAR(n) padding: CHAR(2) lưu 'VI' nhưng 'VN ' (trailing space) — dùng VARCHAR(2) hoặc CHAR(2).
    Thực ra PostgreSQL CHAR và VARCHAR có performance tương đương nhau.

! NULL columns trong PostgreSQL: mỗi column có 1 bit trong NULL bitmap.
    8 nullable columns = 1 byte overhead. Với 100+ columns NULL overhead đáng kể.
    Dùng NOT NULL với DEFAULT khi có thể.

! SMALLINT range: -32,768 đến 32,767. ID counters sẽ overflow nhanh — chỉ dùng cho
    domain-bounded values (age, score, level), KHÔNG dùng cho IDs.
```

---

9. Interning & Deduplication Patterns

Tag: I

Khái niệm

String interning: lưu mỗi unique string 1 lần, sau đó reference bằng ID nguyên. Áp dụng khi có nhiều bản sao của cùng 1 string.

Use case 1: User-Agent string deduplication

```python
# Problem: 10M requests, 100K unique user-agents, mỗi user-agent ~100 bytes
# Naive: lưu raw user-agent mỗi log record = 10M × 100 bytes = 1 GB
# Interning: lưu 1 lần, reference bằng integer ID

# Symbol table trong Redis
def intern_string(value: str) → int:
    """Return integer ID cho string, tạo mới nếu chưa có."""
    # Check if exists
    existing = r.hget("string_table:map", value)
    if existing:
        return int(existing)

    # Allocate new ID
    new_id = r.incr("string_table:counter")

    # Bidirectional mapping
    r.hset("string_table:map", value, new_id)       # string → id
    r.hset("string_table:ids", new_id, value)       # id → string

    return new_id

def lookup_string(string_id: int) → str:
    return r.hget("string_table:ids", string_id).decode()

# Lưu log với interned user-agent
ua_id = intern_string("Mozilla/5.0 (iPhone; CPU iPhone OS 17_4 like Mac OS X)...")
log_entry = {"ts": timestamp, "user_id": 12345, "ua_id": ua_id}  # 4 bytes vs 100 bytes

# Memory savings:
# 10M log records × 4 bytes (int32 ID) = 40 MB
# vs 10M records × 100 bytes (raw string) = 1 GB
# + 100K unique strings × 100 bytes = 10 MB (symbol table)
# Total với interning: 50 MB vs 1 GB → tiết kiệm 20x
```

Use case 2: Dictionary encoding cho analytics

```python
# Columnar analytics: lưu city names, country codes, device types
# dưới dạng integer dictionary-encoded

import pandas as pd

# Raw data: 1M events với city string
```

```
events_raw = pd.DataFrame({
    'city': ['Ho Chi Minh'] * 400_000 + ['Hanoi'] * 300_000 + ['Da Nang'] * 300_000,
    'user_id': range(1_000_000),
    'action': ['click'] * 1_000_000,
})

print(events_raw.memory_usage(deep=True).sum() / 1e6, "MB")
# → ~80 MB (strings stored repeatedly)

# Dictionary encode với pandas Categorical
events_encoded = events_raw.copy()
events_encoded['city'] = events_encoded['city'].astype('category')
events_encoded['action'] = events_encoded['action'].astype('category')

print(events_encoded.memory_usage(deep=True).sum() / 1e6, "MB")
# → ~8 MB (int8 codes + small category dict)
# → Tiết kiệm 10x
```

Use case 3: PostgreSQL lookup table pattern

```
-- Thay vì lưu "Ho Chi Minh" trong mỗi row của 10M rows:
CREATE TABLE cities (
    id   SMALLINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name VARCHAR(100) NOT NULL UNIQUE
);

CREATE TABLE events (
    id      BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    city_id SMALLINT REFERENCES cities(id),  -- 2 bytes vs 20+ bytes
    -- ...
);

-- 10M events × 2 bytes (city_id) = 20 MB
-- vs 10M events × 20 bytes (city_name VARCHAR) = 200 MB
-- + cities table: 200 cities × 100 bytes = 20 KB (negligible)
-- → Tiết kiệm 180 MB
```

Gotchas — Interning & Deduplication

```
Intern table trở thành single point of contention nếu write-heavy.
   Dùng read-through cache: check local cache → Redis → miss → atomic SETNX.

Symbol table size phải bounded. Nếu data cardinality cao (log IDs, timestamps),
   interning không hiệu quả — overhead table còn lớn hơn savings.
   Rule: intern chỉ khi unique values < 1% total records.

Integer IDs thay thế strings làm debugging khó hơn.
   Luôn có API để resolve: id → original string. Log cả 2 trong dev environment.

Pandas Categorical: thêm operation đôi khi không hoạt động với categorical dtype.
   Khi cần merge/join nhiều DataFrames, ensure consistent category mapping.
```

Cheat Sheet tổng hợp

Structure	Bộ nhớ	Use case	So sánh naive	Gotcha chính
Redis Bitmap	122 KB / 1M users	Online status, DAU, retention	50 MB → 122 KB (400x)	Sparse UID → waste
HyperLogLog	12 KB bất kể cardinality	Unique visitor count	50 MB → 12 KB (4000x)	Không list được elements

Structure	Bộ nhớ	Use case	So sánh naive	Gotcha chính
Bloom Filter	1.15 MB / 1M elements 1% FP	Seen notification, URL dedup	70 MB → 1.15 MB (60x)	False positive tồn tại
Count-Min Sketch	53 KB / 10M users	Rate limit, trending, heavy hitter	500 MB → 53 KB (10000x)	Over-estimate only
Roaring Bitmap	~2 KB / 1000 users sparse	Sparse UID sets, search posting lists	1.25 MB → 2 KB (600x)	Không native Redis
BITFIELD	1 MB / 1M users (u8)	Rate counters, level, A/B flags	50 MB → 1 MB (50x)	Overflow policy quan trọng
Trie/Radix	Phụ thuộc prefix sharing	IP lookup, autocomplete, routing	O(n) scan → O(key_len)	Complex impl, use libs
PG compact types	2 MB vs 8 MB / 1M rows	SMALLINT, ENUM, BIT flags	4-10x savings	SMALLINT overflow risk
String interning	50 MB vs 1 GB / 10M logs	User-agent, city, device type	20x savings	Intern table contention

Decision flowchart

```
Cần track Boolean per user?
  → YES: Redis Bitmap (nếu UID sequential) hoặc Roaring Bitmap (sparse UID)

Cần đếm unique elements (cardinality)?
  → YES: Redis HyperLogLog

Cần check "đã thấy/tồn tại chưa"?
  → YES: Bloom Filter (no delete) hoặc Cuckoo Filter (need delete)

Cần đếm tần suất / rate limit?
  → YES: Count-Min Sketch + Top-K

Cần nhiều small counters per user?
  → YES: Redis BITFIELD packed array

Cần prefix matching / autocomplete?
  → YES: Trie hoặc Radix Tree

Lưu string lặp đi lặp lại nhiều lần?
  → YES: String interning / Dictionary encoding
```

File tiếp theo: `optimization-02-kernel-os-tuning.md`

17

Kernel & OS Tuning — Biến VPS 4GB thành cỗ máy chịu 1M connections

> Mục tiêu: Squeeze tối đa hiệu năng từ server nhỏ. Mỗi section đều có math thực tế, giá trị sysctl cụ thể, và lý do production.

File Descriptors B

Tại sao quan trọng

Linux kernel đại diện mọi thứ bằng file descriptor: socket, file, pipe, epoll instance. Mỗi TCP connection = 1 FD. Muốn chịu 1M connections thì cần ít nhất 1M FDs mở cùng lúc.

Default ulimit của hầu hết distro: 1024.

Nghĩa là Nginx mặc định chỉ phục vụ được ~1020 connections (trừ 3 FD chuẩn: stdin/stdout/stderr + một vài internal). Cần tăng lên.

Các tầng giới hạn FD

Linux có 3 tầng limit chồng lên nhau:

```
Tầng 1: fs.file-max          — system-wide ceiling (kernel)
Tầng 2: fs.nr_open           — per-process ceiling (kernel)
Tầng 3: ulimit -n / LimitNOFILE — per-process limit (user/systemd)
```

Tất cả 3 tầng phải được tăng. Thiếu một tầng là fail.

Cấu hình thực tế

Bước 1: Tăng system-wide limit

```
# /etc/sysctl.conf hoặc /etc/sysctl.d/99-production.conf
fs.file-max = 2097152       # 2M FDs cho toàn system
fs.nr_open = 1048576        # 1M FDs max per-process
```

```
sysctl -p  # Apply ngay không cần reboot
```

Bước 2: Tăng per-user limit

```
# /etc/security/limits.conf
*         soft    nofile    1048576
*         hard    nofile    1048576
root      soft    nofile    1048576
root      hard    nofile    1048576
```

Bước 3: Tăng cho systemd services (quan trọng nếu dùng systemd)

```
# /etc/systemd/system/nginx.service.d/override.conf
# Hoặc edit trực tiếp service file
[Service]
LimitNOFILE=1048576
```

```
systemctl daemon-reload
systemctl restart nginx
```

Kiểm tra đã apply chưa:

```
# Check system-wide
cat /proc/sys/fs/file-max
cat /proc/sys/fs/nr_open

# Check process cụ thể (thay PID)
cat /proc/$(pgrep nginx | head -1)/limits | grep "open files"

# Check current usage
cat /proc/sys/fs/file-nr   # Format: used  freed  max
```

Docker containers — gotcha quan trọng

> Docker containers **inherit ulimit từ daemon**, không từ `/etc/security/limits.conf` của host.

```
# /etc/docker/daemon.json
{
  "default-ulimits": {
    "nofile": {
      "Name": "nofile",
      "Hard": 1048576,
      "Soft": 1048576
    }
  }
}
```

Hoặc per-container:

```
docker run --ulimit nofile=1048576:1048576 nginx
```

```
# docker-compose.yml
services:
  nginx:
    ulimits:
      nofile:
        soft: 1048576
        hard: 1048576
```

TCP Stack Tuning cho C1M (1 Million Connections)

Backlog queues — thứ chết đầu tiên khi traffic spike

```
Client SYN  →  [SYN queue]  →  3-way handshake  →  [Accept queue]  →  app accept()
```

Default values gây thảm họa:

- `net.core.somaxconn = 128` — chỉ 128 connections có thể đứng chờ trong accept queue
- `net.ipv4.tcp_max_syn_backlog = 256` — SYN queue tương tự

Khi queue đầy, kernel **silently drops** incoming SYN packets. Client retry sau 1-3 giây. User thấy lag.

Cấu hình đúng:

```
net.core.somaxconn = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.core.netdev_max_backlog = 65535   # NIC packet queue trước khi kernel xử lý
```

> Nginx và các app server cũng cần tăng listen backlog: `nginx listen 80 backlog=65535;`

Ephemeral ports — bottleneck ít ai để ý

Khi server làm **reverse proxy** (Nginx → backend, HAProxy → upstream), mỗi outbound connection cần một ephemeral port. Default range:

```
cat /proc/sys/net/ipv4/ip_local_port_range
# 32768   60999  → chỉ 28231 ports
```

Với 28K ports và mỗi connection mất 60 giây TIME_WAIT, throughput max là:

```
28231 ports / 60 seconds = ~470 new connections/second
```

Fix:

```
net.ipv4.ip_local_port_range = 1024 65535   # 64511 ports available
net.ipv4.tcp_tw_reuse = 1                    # Reuse TIME_WAIT sockets (critical)
net.ipv4.tcp_fin_timeout = 15               # Giảm từ 60s xuống 15s
```

Sau khi fix: `64511 / 15s = ~4300 new connections/second` — tăng 9x.

TCP keepalive — detect dead connections

Default `tcp_keepalive_time = 7200` (2 giờ!). Nghĩa là connection zombie ngồi giữ FD 2 tiếng trước khi bị phát hiện.

```
net.ipv4.tcp_keepalive_time = 300     # Gửi keepalive probe sau 5 phút idle
net.ipv4.tcp_keepalive_intvl = 30    # Probe lại mỗi 30 giây
net.ipv4.tcp_keepalive_probes = 5    # Drop sau 5 lần fail (5×30s = 2.5 phút)
```

Tổng thời gian detect dead connection: `300 + (5 × 30) = 450 giây` thay vì 2 giờ.

Math thực tế cho 1M connections

```
1 TCP socket kernel memory ≈ 3.5KB (sk_buff + socket struct + overhead)
1,000,000 sockets × 3.5KB = 3,500MB = 3.4GB

VPS 4GB RAM:
- Kernel overhead: ~200MB
- TCP sockets 1M: ~3,400MB
- App còn lại: ~400MB   ← chỉ đủ cho app rất nhỏ!

Thực tế khuyến nghị: target 500K connections trên 4GB, hoặc upgrade RAM.
```

Full sysctl TCP block

```
# /etc/sysctl.d/99-tcp-c1m.conf

# Backlog queues
net.core.somaxconn = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.core.netdev_max_backlog = 65535

# Ephemeral ports + TIME_WAIT
net.ipv4.ip_local_port_range = 1024 65535
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_fin_timeout = 15

# Keepalive
net.ipv4.tcp_keepalive_time = 300
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_keepalive_probes = 5

# SYN cookies (chống SYN flood khi queue đầy)
net.ipv4.tcp_syncookies = 1
```

TCP Performance ⓵

TCP Fast Open — save 1 RTT

Bình thường, mỗi TCP connection mới cần 3-way handshake (1.5 RTT) trước khi gửi data. TFO cho phép gửi data **cùng với SYN packet**.

```
Không TFO:  SYN → SYN-ACK → ACK → Request → Response   (1.5 RTT + request time)
Có TFO:     SYN+Data → SYN-ACK+Response                 (0.5 RTT + request time)
```

```
net.ipv4.tcp_fastopen = 3   # 1=client only, 2=server only, 3=both
```

Nginx config:

```
listen 443 ssl http2 fastopen=256 reuseport;
```

TCP Window Scaling và Buffers

```
net.ipv4.tcp_window_scaling = 1     # Cho phép window > 64KB (cần cho high-latency links)

# Receive/Send buffers (min default max)
net.core.rmem_max = 16777216        # 16MB max receive buffer per socket
net.core.wmem_max = 16777216        # 16MB max send buffer per socket
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216

# Auto-tuning (kernel tự điều chỉnh buffer)
net.ipv4.tcp_moderate_rcvbuf = 1
```

Math buffer:

```
Default rmem_default = 87380 bytes ≈ 85KB
Với latency 100ms (cross-region), bandwidth-delay product:
  Throughput = Window / RTT = 85KB / 0.1s = 850KB/s per connection

Với 16MB buffer:
  Throughput = 16MB / 0.1s = 160MB/s per connection  — đủ cho 10Gbps links
```

BBR Congestion Control A

BBR (Bottleneck Bandwidth and Round-trip propagation time) thay thế CUBIC mặc định. BBR mô hình hóa network thay vì phản ứng với packet loss như CUBIC.

Lợi thế của BBR: - Throughput cao hơn 2-25x trên lossy networks (>1% packet loss) - Latency thấp hơn vì không cần fill buffer để đo bandwidth - Phù hợp cho VPS cloud thường có shared network với noise

```
net.ipv4.tcp_congestion_control = bbr
net.core.default_qdisc = fq        # Fair Queuing — cần pair với BBR
```

Kiểm tra BBR available:

```
sysctl net.ipv4.tcp_available_congestion_control
# Output phải có: bbr
# Nếu không có: cần kernel 4.9+
uname -r  # Check kernel version
```

BBR cần kernel 4.9+. Ubuntu 18.04+ và CentOS 8+ đều đủ điều kiện. CentOS 7 default kernel 3.10 → cần upgrade.

Memory Management I

Swap và vm.swappiness

```
vm.swappiness = 10   # Kernel sẽ swap khi còn 10% RAM free (default 60)
```

Logic: Server network-heavy như Nginx/Redis không muốn bị swap. Swap gây latency spike đột ngột. Giảm swappiness giữ hot data trong RAM lâu hơn.

> Nếu dùng Redis: một số người recommend `vm.swappiness = 0` nhưng điều này có thể gây OOM kill thay vì swap khi memory pressure. `swappiness = 1` là compromise tốt.

vm.overcommit_memory — cần thiết cho Redis

```
vm.overcommit_memory = 1   # Allow overcommit (default 0 = heuristic)
```

Redis `BGSAVE` và `BGREWRITEAOF` dùng `fork()`. Khi fork, kernel cần "commit" đủ memory cho child process (copy-on-write). Nếu `overcommit_memory = 0`, Redis có thể fail fork với lỗi:

```
Can't save in background: fork: Cannot allocate memory
```

`vm.overcommit_memory = 1` cho phép kernel overcommit — dùng được vì CoW đảm bảo memory thực sự không được allocate cho đến khi write.

Dirty page write-back

```
vm.dirty_ratio = 20                # Bắt đầu synchronous write khi 20% RAM là dirty pages
vm.dirty_background_ratio = 5     # Background write-back bắt đầu ở 5%
vm.dirty_writeback_centisecs = 500 # Flush dirty pages mỗi 5 giây
vm.dirty_expire_centisecs = 3000   # Pages dirty > 30s phải được flush
```

Cho database servers: Giảm dirty_ratio để tránh latency spike lớn khi flush:

```
vm.dirty_ratio = 5
vm.dirty_background_ratio = 2
```

Transparent Huge Pages — BẮT BUỘC disable cho Redis/MongoDB !

THP (Transparent Huge Pages) group các 4KB pages thành 2MB pages để giảm TLB pressure. Nghe có vẻ tốt, nhưng:

- Khi kernel cần "compact" memory để tạo 2MB contiguous page, nó **stop-the-world** briefly
- Redis, MongoDB, PostgreSQL đều có latency spike do THP compaction
- Production outages thực tế đã xảy ra vì THP

```
# Disable ngay lập tức (không cần reboot)
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag

# Persist qua reboot — thêm vào /etc/rc.local hoặc systemd service
```

Systemd service để persist:

```
# /etc/systemd/system/disable-thp.service
[Unit]
Description=Disable Transparent Huge Pages
After=network.target

[Service]
Type=oneshot
ExecStart=/bin/sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/enabled'
ExecStart=/bin/sh -c 'echo never > /sys/kernel/mm/transparent_hugepage/defrag'
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
```

OOM Killer Tuning A

Khi RAM hết, Linux OOM Killer chọn process để kill. Default nó có thể kill Nginx hoặc Postgres thay vì process rác.

```
# Protect critical process (giá trị từ -1000 đến 1000)
# -1000 = never kill, 1000 = kill first

# Protect Postgres
echo -500 > /proc/$(pgrep postgres | head -1)/oom_score_adj

# Hoặc trong systemd service:
# OOMScoreAdjust=-500
```

I/O Optimization I

I/O Scheduler

Linux I/O scheduler quyết định thứ tự đọc/ghi disk. Với SSD/NVMe, scheduler không cần reorder I/O (không có seek time).

```
# Check scheduler hiện tại
cat /sys/block/sda/queue/scheduler

# Set none/noop cho SSD (bypass scheduler)
```

```
echo none > /sys/block/sda/queue/scheduler    # Kernel 5.x
echo noop > /sys/block/sda/queue/scheduler    # Kernel 4.x

# Persist qua reboot (udev rule)
# /etc/udev/rules.d/60-scheduler.rules
ACTION=="add|change", KERNEL=="sd[a-z]|nvme[0-9]n[0-9]", \
  ATTR{queue/rotational}=="0", \
  ATTR{queue/scheduler}="none"
```

Với HDD: Dùng `mq-deadline` hoặc `bfq` để merge sequential I/O.

Read-ahead

```
# Check current read-ahead
blockdev --getra /dev/sda   # Output: sectors (512 bytes each)

# Giảm read-ahead cho random I/O (database)
blockdev --setra 256 /dev/sda   # 256 sectors = 128KB

# Tăng read-ahead cho sequential I/O (log files, backup)
blockdev --setra 4096 /dev/sda  # 4096 sectors = 2MB
```

noatime mount option — giảm 20-30% write I/O

Mặc định Linux update `atime` (access time) mỗi lần đọc file. Điều này biến mọi read thành write lên disk.

```
# /etc/fstab
UUID=xxx  /  ext4  defaults,noatime,nodiratime  0 1
```

`noatime` disable hoàn toàn. `relatime` (default trên nhiều distro) chỉ update khi atime < mtime — compromise tốt hơn.

io_uring — async I/O thế hệ mới A

`io_uring` (Linux 5.1+) cung cấp async I/O interface zero-copy với ring buffer shared giữa kernel và userspace. Throughput cao hơn `aio` cũ 2-3x.

```
# Check io_uring available
ls /sys/kernel/debug/io_uring 2>/dev/null && echo "available"

# Nginx hỗ trợ io_uring từ 1.25.0+
# Thêm vào nginx.conf
events {
    use io_uring;
    worker_connections 65535;
}
```

Epoll & Event-driven I/O I

Tại sao epoll thống trị

```
Mechanism   Time Complexity   Max FDs    Behavior
select      O(n)              1024       Copy FD set mỗi call, scan toàn bộ
poll        O(n)              ~unlimited  Copy FD list mỗi call, scan toàn bộ
epoll       O(1)              ~unlimited  Kernel notify only ready FDs
```

Với 10,000 connections: - `select/poll`: mỗi event phải scan 10,000 FDs → 10,000 operations - `epoll`: kernel chỉ trả về FDs có event → thường 1-10 operations

Epoll là lý do Nginx có thể handle hàng triệu connections với 1 worker.

epoll internals A

```
// Tạo epoll instance
int epfd = epoll_create1(0);

// Thêm FD vào watch list
struct epoll_event ev;
ev.events = EPOLLIN | EPOLLET;  // Edge-triggered
ev.data.fd = sockfd;
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &ev);

// Wait for events (block tối đa timeout ms)
int nfds = epoll_wait(epfd, events, MAX_EVENTS, timeout);
```

Level-triggered vs Edge-triggered:

```
Level-triggered (LT): epoll_wait trả về khi FD CÒN data chưa đọc
  → Safe, dễ code, có thể partial read
  → Nếu không đọc hết, tiếp tục được notify

Edge-triggered (ET): epoll_wait trả về chỉ khi có DATA MỚI arrive
  → Phải đọc hết buffer trong 1 lần (loop until EAGAIN)
  → Ít syscalls hơn = performance cao hơn
  → Nếu bỏ sót data, sẽ không nhận notification nữa → connection stall
```

Nginx dùng edge-triggered với non-blocking sockets.

SO_REUSEPORT — scale Nginx workers I

```
# /etc/sysctl.conf không cần — là socket option
# Nginx config:
events {
    worker_connections 65535;
}

http {
    server {
        listen 80 reuseport;
        listen 443 ssl reuseport;
    }
}
```

Với `reuseport`, mỗi Nginx worker process bind port 80 riêng. Kernel load-balances incoming connections giữa các workers. Không có master process bottleneck.

! SO_REUSEPORT + connection migration trên kernel < 4.6: Khi worker restart, in-flight connections có thể drop. Kernel 4.6+ fix điều này với `EBPF` socket migration. Trên kernel cũ, cần graceful reload thay vì hard restart.

```
nginx -s reload   # Graceful, không drop connections
systemctl restart nginx  # Hard restart, có thể drop connections
```

Process & Thread Limits B

```
# Tăng max PIDs (default 32768, max 4194304)
kernel.pid_max = 4194304

# Max threads system-wide
kernel.threads-max = 4194304
```

```
# Max processes per user
# /etc/security/limits.conf
*    soft    nproc    65535
*    hard    nproc    65535
```

Cho containers (cgroups):

```
# Giới hạn CPU cho container
# /sys/fs/cgroup/cpu/docker/<container_id>/cpu.shares = 1024  (relative weight)
# /sys/fs/cgroup/cpu/docker/<container_id>/cpu.cfs_quota_us = 200000  (200ms/100ms = 2 cores)

# Giới hạn memory
# /sys/fs/cgroup/memory/docker/<container_id>/memory.limit_in_bytes
```

Practical Profiles — Sysctl Templates

Profile: Web Server (Nginx/HAProxy)

```
# /etc/sysctl.d/99-webserver.conf

# File descriptors
fs.file-max = 2097152
fs.nr_open = 1048576

# TCP backlog
net.core.somaxconn = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.core.netdev_max_backlog = 65535

# Ports + TIME_WAIT
net.ipv4.ip_local_port_range = 1024 65535
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_fin_timeout = 15

# Keepalive
net.ipv4.tcp_keepalive_time = 300
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_keepalive_probes = 5

# Performance
net.ipv4.tcp_fastopen = 3
net.ipv4.tcp_window_scaling = 1
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
net.ipv4.tcp_congestion_control = bbr
net.core.default_qdisc = fq

# Security
net.ipv4.tcp_syncookies = 1

# Memory
vm.swappiness = 10
```

Profile: Database (PostgreSQL/MySQL)

```
# /etc/sysctl.d/99-database.conf

# Shared memory cho PostgreSQL
kernel.shmmax = 68719476736      # 64GB max shared memory segment
kernel.shmall = 4294967296       # Total shared pages
```

```
# Memory
vm.swappiness = 5                # Database cực kỳ nhạy cảm với swap
vm.dirty_ratio = 5               # Flush dirty pages sớm
vm.dirty_background_ratio = 2
vm.overcommit_memory = 0         # Database không cần overcommit

# Huge pages cho PostgreSQL (optional, significant benefit)
vm.nr_hugepages = 512            # 512 × 2MB = 1GB huge pages
# PostgreSQL: huge_pages = on trong postgresql.conf

# I/O
vm.dirty_writeback_centisecs = 100   # Flush mỗi 1 giây (database cần durability)

# Network (cho replica sync)
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

# File descriptors
fs.file-max = 2097152
```

Profile: Cache (Redis/Memcached)

```
# /etc/sysctl.d/99-cache.conf

# Redis requirements
vm.overcommit_memory = 1         # Cần cho BGSAVE fork
vm.swappiness = 1                # Tránh swap bằng mọi giá

# THP phải disable riêng (không phải sysctl)
# echo never > /sys/kernel/mm/transparent_hugepage/enabled

# Network
net.core.somaxconn = 65535
net.ipv4.tcp_tw_reuse = 1
net.ipv4.ip_local_port_range = 1024 65535

# Memory
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

# File descriptors (Redis default maxclients = 10000)
fs.file-max = 2097152
```

Profile: Queue (RabbitMQ/Kafka)

```
# /etc/sysctl.d/99-queue.conf

# File descriptors — Kafka cần nhiều FD cho log segments
fs.file-max = 2097152
fs.nr_open = 1048576

# Network buffers — Kafka throughput-sensitive
net.core.rmem_max = 134217728     # 128MB
net.core.wmem_max = 134217728
net.ipv4.tcp_rmem = 4096 65536 134217728
net.ipv4.tcp_wmem = 4096 65536 134217728

# TCP
net.core.somaxconn = 65535
net.ipv4.tcp_tw_reuse = 1
net.ipv4.ip_local_port_range = 1024 65535

# I/O — Kafka là log-sequential, read-ahead cao
# blockdev --setra 4096 /dev/sda  (set riêng, không phải sysctl)

# Memory
vm.swappiness = 1
```

```
vm.dirty_ratio = 20              # Kafka batch writes, cho phép accumulate
vm.dirty_background_ratio = 10
```

Apply & Verify

```
# Apply tất cả sysctl files
sysctl --system

# Verify các giá trị quan trọng
sysctl net.core.somaxconn
sysctl net.ipv4.tcp_tw_reuse
sysctl fs.file-max
sysctl net.ipv4.tcp_congestion_control

# Monitor connections realtime
ss -s                            # Summary stats
ss -tn state time-wait | wc -l   # Count TIME_WAIT
watch -n1 'ss -s'

# Monitor FD usage
cat /proc/sys/fs/file-nr         # used / freed / max
lsof -n | wc -l                  # Total open FDs across all processes

# Stress test với wrk
wrk -t12 -c400 -d30s http://localhost/
```

Quick Reference Checklist

```
[ ] fs.file-max và fs.nr_open đã tăng
[ ] /etc/security/limits.conf đã update
[ ] systemd LimitNOFILE cho mỗi service đã set
[ ] Docker daemon.json đã có default-ulimits
[ ] net.core.somaxconn = 65535
[ ] net.ipv4.tcp_tw_reuse = 1
[ ] net.ipv4.ip_local_port_range mở rộng
[ ] BBR congestion control (nếu kernel 4.9+)
[ ] THP disabled cho Redis/MongoDB/PostgreSQL
[ ] vm.overcommit_memory = 1 cho Redis
[ ] I/O scheduler = none cho SSD
[ ] noatime trong /etc/fstab
[ ] Nginx listen với reuseport và backlog=65535
```

18

Connection Pooling & Management — Multiplexing 1M users qua 100 connections

> Core insight: App không cần 1M database connections. Pool và multiplex. 1M users → vài nghìn app connections → 50-200 DB connections.

The Problem — Tại sao connections là bottleneck B

PostgreSQL: mỗi connection = 1 process

PostgreSQL dùng **process-per-connection** model (không phải thread). Mỗi connection fork ra 1 backend process.

```
Memory per PostgreSQL backend process:
  - Shared memory (shared_buffers): shared, không tính riêng
  - Per-process: ~5-10MB (work_mem, stack, overhead)
  - Thực tế: 7-12MB mỗi connection

100 connections  = ~700MB - 1.2GB RAM chỉ cho PG backends
1000 connections = ~7GB - 12GB RAM → OOM trên hầu hết servers
```

Thêm vào đó: PostgreSQL có `max_connections` default = 100. Mỗi connection tốn slot trong shared memory ngay cả khi idle.

MySQL/MariaDB: mỗi connection = 1 thread

Thread nhẹ hơn process, nhưng:

```
Memory per MySQL thread:
  - thread_stack: 256KB (default)
  - read_buffer_size + sort_buffer_size + join_buffer_size: ~2-8MB khi active
  - Overhead: ~1MB

1000 connections × 1MB = 1GB RAM khi idle
1000 connections × 8MB = 8GB RAM khi all active
```

MySQL `max_connections` default = 151. Vượt quá → "Too many connections" error.

Redis: connection overhead nhỏ nhưng vẫn có giới hạn

Redis single-threaded xử lý commands, nhưng mỗi connection: - 1 FD (file descriptor) - ~20KB memory cho client buffer

```
10,000 connections × 20KB = 200MB — chấp nhận được
Nhưng: Redis mặc định maxclients = 10000, FD limit cần tương ứng
```

Math tổng quan: tại sao cần pooling

```
Scenario: 10,000 concurrent users, mỗi request cần 1 DB query

Without pooling:
  10,000 app connections → 10,000 PostgreSQL connections
  10,000 × 10MB = 100GB RAM chỉ cho PG backends → IMPOSSIBLE

With PgBouncer (transaction pooling):
  10,000 app connections → PgBouncer → 50 PostgreSQL connections
  50 × 10MB = 500MB RAM cho PG backends → FEASIBLE

Multiplier: 10,000 / 50 = 200x reduction
```

PgBouncer — Connection Pooler cho PostgreSQL [B/I]

Các chế độ pooling

PgBouncer có 3 pooling modes, khác nhau ở khi nào connection được trả về pool:

```
Mode              Connection hold time      Use case
──────────────────────────────────────────────────────────────────
session           Suốt session lifetime     Cần session state (LISTEN, SET, temp tables)
transaction       Chỉ trong transaction     95% use cases — recommended default
statement         Mỗi statement             Rất hạn chế, không hỗ trợ multi-statement TX
```

Transaction pooling — cơ chế hoạt động:

```
Client A:  BEGIN → query → COMMIT  [connection trả về pool]
Client B:                           [lấy connection từ pool]  → BEGIN → ...
Client A:  [request mới]            [lấy connection từ pool]  → BEGIN → ...
```

Kết quả: 1000 app clients chỉ cần 50 PostgreSQL connections nếu average transaction < 50ms.

Cài đặt và cấu hình

```
# Ubuntu/Debian
apt install pgbouncer

# CentOS/RHEL
yum install pgbouncer
```

`/etc/pgbouncer/pgbouncer.ini`:

```
[databases]
# Format: dbname = host=... port=... dbname=...
myapp = host=127.0.0.1 port=5432 dbname=myapp_production
myapp_readonly = host=replica-host port=5432 dbname=myapp_production

[pgbouncer]
listen_addr = 127.0.0.1
listen_port = 6432

# Auth
auth_type = hba                    # Dùng pg_hba.conf của PostgreSQL
auth_file = /etc/pgbouncer/userlist.txt

# Pooling mode
pool_mode = transaction

# Pool sizing
default_pool_size = 25             # Connections tới PG per database+user pair
```

```
min_pool_size = 5                   # Giữ tối thiểu 5 connections sẵn sàng
reserve_pool_size = 5               # Extra connections khi load spike
reserve_pool_timeout = 3            # Chờ 3s trước khi dùng reserve pool

# Client limits
max_client_conn = 10000             # Max app connections vào PgBouncer
client_idle_timeout = 600           # Disconnect idle clients sau 10 phút

# Server connections
server_idle_timeout = 600           # Close idle PG connections sau 10 phút
server_lifetime = 3600              # Force reconnect sau 1 giờ
server_reset_query = DISCARD ALL    # Reset session state sau mỗi transaction

# Logging
logfile = /var/log/pgbouncer/pgbouncer.log
pidfile = /var/run/pgbouncer/pgbouncer.pid

# Admin
admin_users = pgbouncer_admin
stats_users = monitoring_user
```

`/etc/pgbouncer/userlist.txt`:

```
# Format: "username" "md5hash" hoặc "username" "plain_password"
"myapp_user" "md5abc123..."
"pgbouncer_admin" "admin_password"
```

Sizing formula

```
Optimal pool_size = Ncores × 2 + Nspindles

Ví dụ server 4 CPU, SSD (1 effective spindle):
  pool_size = 4 × 2 + 1 = 9   → round lên 10-15

Ví dụ server 8 CPU, RAID-10 với 4 drives:
  pool_size = 8 × 2 + 4 = 20  → round lên 20-25

Rule of thumb: bắt đầu với 2-4x số CPU, benchmark, tune
```

Common mistake: Set pool_size = 200 vì muốn "an toàn". Thực ra database thrash vì quá nhiều concurrent queries tranh CPU. Ít connections hơn, queue trong PgBouncer, throughput thực tế cao hơn.

PgBouncer limitations — PHẢI biết trước khi dùng

Prepared statements không work trong transaction mode:

```
-- Cái này sẽ FAIL với transaction pooling
PREPARE stmt AS SELECT * FROM users WHERE id = $1;
EXECUTE stmt(123);
```

Fix: Dùng unnamed prepared statements (nhiều ORM tự xử lý), hoặc set `server_reset_query = DISCARD ALL`.

Node.js với `pg` library: pass `prepare: false` khi connect qua PgBouncer.

SET commands lost giữa transactions:

```
-- Transaction 1
SET search_path = myschema;  -- OK trong transaction này
COMMIT;

-- Transaction 2 (có thể trên connection khác)
SELECT * FROM mytable;  -- search_path đã reset!
```

Fix: Set `search_path` trong PostgreSQL user config hoặc database config, không phải per-session.

LISTEN/NOTIFY cần session mode:

```
LISTEN không tương thích với transaction pooling.
Dùng connection riêng với session mode cho LISTEN/NOTIFY,
hoặc dùng PgBouncer với mixed pools (cấu hình 2 database entries).
```

Monitoring và debugging

```
-- Connect vào PgBouncer admin console
psql -h 127.0.0.1 -p 6432 -U pgbouncer_admin pgbouncer

-- Xem pool stats
SHOW POOLS;
-- Output: database, user, cl_active, cl_waiting, sv_active, sv_idle, sv_used, maxwait

-- Xem client connections
SHOW CLIENTS;

-- Xem server connections (tới PostgreSQL)
SHOW SERVERS;

-- Stats tổng quan
SHOW STATS;

-- Key metrics cần monitor:
-- cl_waiting > 0: clients đang chờ connection → tăng pool_size
-- maxwait > 1000ms: nghiêm trọng, tăng pool_size hoặc tăng PG connections
-- sv_idle = 0: pool exhausted liên tục → pool_size quá nhỏ
```

ProxySQL — Connection Pooler cho MySQL

Kiến trúc và tính năng

ProxySQL là layer 7 proxy cho MySQL với: - Connection multiplexing: 10K app connections → 100 MySQL connections - Query routing: read/write split tự động - Query caching - Query rewriting - Failover tự động

```
# Cài đặt
apt install proxysql   # Ubuntu
# Hoặc download từ proxysql.com

# Admin interface trên port 6032
mysql -u admin -padmin -h 127.0.0.1 -P 6032
```

Cấu hình cơ bản

```
-- Thêm backend servers
INSERT INTO mysql_servers (hostgroup_id, hostname, port, max_connections)
VALUES
  (1, 'primary-db', 3306, 100),    -- hostgroup 1 = writers
  (2, 'replica-db', 3306, 200);    -- hostgroup 2 = readers

-- Thêm users
INSERT INTO mysql_users (username, password, default_hostgroup)
VALUES ('myapp', 'password', 1);   -- Default: writes go to primary

-- Query routing rules (read/write split)
INSERT INTO mysql_query_rules (rule_id, active, match_pattern, destination_hostgroup, apply)
VALUES
  (1, 1, '^SELECT.*FOR UPDATE', 1, 1),  -- SELECT FOR UPDATE → primary
  (2, 1, '^SELECT', 2, 1);              -- Other SELECTs → replicas

-- Apply config
```

```
LOAD MYSQL SERVERS TO RUNTIME;
LOAD MYSQL USERS TO RUNTIME;
LOAD MYSQL QUERY RULES TO RUNTIME;
SAVE MYSQL SERVERS TO DISK;
SAVE MYSQL USERS TO DISK;
SAVE MYSQL QUERY RULES TO DISK;
```

Monitoring ProxySQL

```
-- Connection pool status
SELECT * FROM stats.stats_mysql_connection_pool;

-- Query digest (top slow queries)
SELECT * FROM stats.stats_mysql_query_digest ORDER BY sum_time DESC LIMIT 10;

-- Key columns trong stats_mysql_connection_pool:
-- ConnUsed: connections đang active
-- ConnFree: connections available trong pool
-- ConnOK: successful connections
-- ConnERR: failed connections
-- Latency_us: latency đến backend (microseconds)
```

Application-Level Pooling ℹ

Node.js — pg (node-postgres)

```
const { Pool } = require('pg');

const pool = new Pool({
  host: '127.0.0.1',
  port: 6432,           // PgBouncer port
  database: 'myapp',
  user: 'myapp_user',
  password: process.env.DB_PASSWORD,

  // Pool config
  max: 20,              // Max connections trong pool
  min: 2,               // Min idle connections
  idleTimeoutMillis: 30000,     // Close idle connection sau 30s
  connectionTimeoutMillis: 2000, // Timeout khi acquire connection

  // Quan trọng khi qua PgBouncer transaction mode
  allowExitOnIdle: true,
});

// Sử dụng
const client = await pool.connect();
try {
  await client.query('BEGIN');
  const result = await client.query('SELECT * FROM users WHERE id = $1', [userId]);
  await client.query('COMMIT');
  return result.rows;
} catch (err) {
  await client.query('ROLLBACK');
  throw err;
} finally {
  client.release();   // CRITICAL: luôn release, kể cả khi có error
}
```

⚠ Pool leak detection:

```
// Log warning nếu pool bị leak
pool.on('error', (err, client) => {
```

```
  console.error('Unexpected error on idle client', err);
});

// Monitor pool stats
setInterval(() => {
  console.log({
    total: pool.totalCount,
    idle: pool.idleCount,
    waiting: pool.waitingCount,
  });
}, 30000);
```

Go — database/sql

```
import (
    "database/sql"
    "time"
    _ "github.com/lib/pq"
)

func NewDB(dsn string) (*sql.DB, error) {
    db, err := sql.Open("postgres", dsn)
    if err != nil {
        return nil, err
    }

    // Pool configuration
    db.SetMaxOpenConns(25)           // Max connections mở cùng lúc
    db.SetMaxIdleConns(10)           // Max idle connections giữ trong pool
    db.SetConnMaxLifetime(5 * time.Minute)   // Force reconnect sau 5 phút
    db.SetConnMaxIdleTime(2 * time.Minute)   // Close idle connection sau 2 phút

    // Verify connection
    if err := db.Ping(); err != nil {
        return nil, err
    }

    return db, nil
}

// Go sql.DB tự quản lý pool — không cần manual connect/release
func GetUser(db *sql.DB, id int) (*User, error) {
    var user User
    err := db.QueryRow("SELECT id, name FROM users WHERE id = $1", id).
        Scan(&user.ID, &user.Name)
    return &user, err
}
```

Sizing Go pool:

```
// Rule: MaxOpenConns = số goroutines có thể concurrent query
// Nếu app có 100 goroutines xử lý requests:
db.SetMaxOpenConns(100)
db.SetMaxIdleConns(25)  // ~25% của max, giữ connections ấm
```

Python — SQLAlchemy

```
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool

engine = create_engine(
    "postgresql+psycopg2://user:pass@localhost:6432/myapp",

    # Pool config
    poolclass=QueuePool,
    pool_size=10,           # Persistent connections trong pool
    max_overflow=20,        # Extra connections khi pool đầy (temporary)
```

```
    pool_recycle=3600,      # Reconnect sau 1 giờ (tránh stale connections)
    pool_pre_ping=True,     # Test connection trước khi dùng (phát hiện disconnect)
    pool_timeout=30,        # Timeout khi chờ lấy connection từ pool

    # Connection args
    connect_args={
        "connect_timeout": 10,
        "options": "-c statement_timeout=30000",  # 30s query timeout
    }
)

# SQLAlchemy session (recommended)
from sqlalchemy.orm import Session

with Session(engine) as session:
    users = session.execute("SELECT * FROM users").fetchall()
    # Connection tự động trả về pool khi exit context manager
```

pool_pre_ping: Thêm 1 lightweight `SELECT 1` trước mỗi query. Tốn ~0.1ms nhưng tránh được lỗi "connection was closed" sau network blip. **Nên bật trong production.**

Java — HikariCP (connection pool tốt nhất cho JVM)

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://localhost:6432/myapp");
config.setUsername("myapp_user");
config.setPassword(System.getenv("DB_PASSWORD"));

// Pool sizing
config.setMaximumPoolSize(25);        // Max connections
config.setMinimumIdle(5);             // Min idle connections
config.setConnectionTimeout(30000);   // 30s timeout để acquire connection
config.setIdleTimeout(600000);        // 10 phút idle → close connection
config.setMaxLifetime(1800000);       // 30 phút max connection lifetime

// Health check
config.setConnectionTestQuery("SELECT 1");
config.setKeepaliveTime(60000);       // Keepalive mỗi 60s

HikariDataSource dataSource = new HikariDataSource(config);
```

HikariCP sizing formula (từ HikariCP docs):

```
pool_size = Tn × (Cm - 1) + 1

Tn = số threads/goroutines tối đa xử lý requests
Cm = số concurrent DB queries mỗi thread cần cùng lúc

Ví dụ: 50 request threads, mỗi thread cần tối đa 2 connections:
  pool_size = 50 × (2 - 1) + 1 = 51

Thực tế: Cm thường = 1 (mỗi thread 1 query tại 1 thời điểm)
  pool_size = 50 × (1 - 1) + 1 = 1   ← quá nhỏ, không thực tế

Practical formula: pool_size = num_cores × 2, không vượt quá 100
```

HTTP Connection Pooling [B/I]

Nginx upstream keepalive — cực kỳ quan trọng

Không có `keepalive`: Nginx tạo TCP connection mới tới backend **cho mỗi request**. 3-way handshake + TLS handshake mỗi lần = 50-200ms overhead mỗi request.

```
upstream backend {
    server 127.0.0.1:8080;
    server 127.0.0.1:8081;

    # Giữ 100 idle connections tới backends
    keepalive 100;
    keepalive_requests 1000;     # Reuse connection tối đa 1000 requests
    keepalive_time 1h;           # Close connection sau 1 giờ
    keepalive_timeout 60s;       # Close idle connection sau 60s
}

server {
    location /api/ {
        proxy_pass http://backend;

        # PHẢI set HTTP/1.1 để keepalive hoạt động
        proxy_http_version 1.1;
        proxy_set_header Connection "";   # Clear Connection header
    }
}
```

Mặc định Nginx dùng HTTP/1.0 cho upstream connections. HTTP/1.0 không support keepalive. Phải set `proxy_http_version 1.1` và `proxy_set_header Connection ""`.

HTTP/2 multiplexing

HTTP/2 cho phép nhiều concurrent requests trên **1 TCP connection** (stream multiplexing).

```
HTTP/1.1 với keepalive:
  Connection 1: Request A → Response A
  Connection 1: Request B → Response B  (sequential)
  Cần 6 connections để có 6 concurrent requests (browser limit)

HTTP/2:
  Connection 1: Stream 1 (Request A) ──────────── Response A
  Connection 1: Stream 2 (Request B) ── Response B
  Connection 1: Stream 3 (Request C) ──────────────── Response C
  1 connection cho unlimited concurrent streams
```

```
# Nginx HTTP/2 config
server {
    listen 443 ssl http2;

    # HTTP/2 push (server → client proactively)
    http2_push_preload on;

    # Tuning
    http2_max_concurrent_streams 128;    # Streams per connection
    http2_chunk_size 8k;
}
```

gRPC connection pooling

gRPC dùng HTTP/2, mỗi Channel = 1 HTTP/2 connection với nhiều streams.

```
// Go gRPC client
import "google.golang.org/grpc"

// Single connection cho simple use case
conn, err := grpc.Dial("localhost:50051",
    grpc.WithTransportCredentials(creds),
    grpc.WithDefaultCallOptions(
        grpc.MaxCallRecvMsgSize(10*1024*1024),  // 10MB
    ),
)
client := pb.NewMyServiceClient(conn)
```

```
// Connection pool cho high throughput
// Dùng nhiều channels
type Pool struct {
    conns  []*grpc.ClientConn
    mu     sync.Mutex
    idx    int
}

func (p *Pool) Get() *grpc.ClientConn {
    p.mu.Lock()
    defer p.mu.Unlock()
    conn := p.conns[p.idx%len(p.conns)]
    p.idx++
    return conn
}
```

Redis Connection Pooling [B/I]

Client libraries tự pool

Hầu hết Redis clients tự quản lý pool. Key config:

Node.js — ioredis:

```
const Redis = require('ioredis');

const redis = new Redis({
  host: '127.0.0.1',
  port: 6379,

  // Pool config
  maxRetriesPerRequest: 3,
  enableReadyCheck: true,

  // Connection pool (ioredis tự quản lý)
  // Default: 1 connection, auto-reconnect

  // Cho cluster:
  // Mỗi node có pool riêng
});

// Lazy connect (default) — connect khi cần
// Hoặc force connect:
await redis.connect();
```

Python — redis-py:

```
import redis

# Connection pool tường minh
pool = redis.ConnectionPool(
    host='127.0.0.1',
    port=6379,
    max_connections=50,
    decode_responses=True,
)

r = redis.Redis(connection_pool=pool)

# Async version
import redis.asyncio as aioredis

pool = aioredis.ConnectionPool.from_url(
    "redis://localhost:6379",
    max_connections=50,
```

```
)
r = aioredis.Redis(connection_pool=pool)
```

Pipelining — batch commands

Pipeline gửi nhiều commands trong 1 network round-trip:

```
# Không pipeline: 3 round-trips
r.set('key1', 'value1')   # RTT 1
r.set('key2', 'value2')   # RTT 2
r.set('key3', 'value3')   # RTT 3

# Với pipeline: 1 round-trip
pipe = r.pipeline()
pipe.set('key1', 'value1')
pipe.set('key2', 'value2')
pipe.set('key3', 'value3')
results = pipe.execute()   # Gửi tất cả cùng 1 lúc

# Performance gain: ~10x throughput vs sequential trên high-latency connection
```

MULTI/EXEC (transaction) trong pipeline:

```
with r.pipeline() as pipe:
    pipe.multi()
    pipe.set('key1', 'value1')
    pipe.incr('counter')
    pipe.execute()
```

SUBSCRIBE cần dedicated connection

```
# SSCRIBE blocks connection — không dùng chung pool connection
pubsub = r.pubsub()
pubsub.subscribe('channel-name')

# Chạy trong thread riêng
import threading

def listener():
    for message in pubsub.listen():
        if message['type'] == 'message':
            handle_message(message['data'])

thread = threading.Thread(target=listener, daemon=True)
thread.start()

# Connection dùng cho commands bình thường vẫn available
r.set('other-key', 'value')   # Dùng pool connection khác
```

Redis Cluster — pool per node

```
from redis.cluster import RedisCluster

rc = RedisCluster(
    host="localhost",
    port=6379,

    # Pool per node
    connection_pool_class_kwargs={
        "max_connections": 50,
    },

    # MOVED/ASK handling
    skip_full_coverage_check=True,
)
```

```
# Redis Cluster tự route keys đến đúng node
rc.set("mykey", "value")   # Tự tính hash slot → node
```

Connection Lifecycle Management I

Idle connection detection

```
# SQLAlchemy pool_pre_ping
engine = create_engine(url, pool_pre_ping=True)
# → SELECT 1 trước mỗi query, recycle nếu connection dead

# PgBouncer: server_check_query = SELECT 1
# Interval: server_check_delay = 30  (seconds)
```

server_lifetime trong PgBouncer:

```
server_lifetime = 3600      # Force reconnect sau 1 giờ
# Tại sao: tránh stale connections, giúp load balancing khi scale PG nodes
```

Graceful draining khi deploy A

Khi deploy mới, cần drain connections đang active trước khi shutdown:

```
# PgBouncer: pause rồi drain
psql -h 127.0.0.1 -p 6432 -U admin pgbouncer -c "PAUSE myapp;"
# Chờ in-flight queries xong
sleep 5
# Deploy service mới
systemctl restart myapp
# Resume
psql -h 127.0.0.1 -p 6432 -U admin pgbouncer -c "RESUME myapp;"
```

Kubernetes graceful shutdown:

```
# Deployment spec
spec:
  template:
    spec:
      terminationGracePeriodSeconds: 30   # Cho 30s drain
      containers:
        - lifecycle:
            preStop:
              exec:
                command: ["/bin/sh", "-c", "sleep 15"]  # Chờ load balancer remove pod
```

Connection leak detection I

Connection leak: code acquire connection nhưng không release → pool dần cạn.

Node.js pg pool:

```
// Timeout khi client không release
const pool = new Pool({
  connectionTimeoutMillis: 2000,
  idleTimeoutMillis: 10000,
});

// Detect leak: log khi pool exhausted
pool.on('connect', (client) => {
  client.on('error', (err) => console.error('client error', err));
});
```

```
// Monitor waiting count
setInterval(() => {
  if (pool.waitingCount > 0) {
    console.warn(`[DB] ${pool.waitingCount} queries waiting for connection!`);
    console.warn(`[DB] Pool: total=${pool.totalCount}, idle=${pool.idleCount}`);
  }
}, 5000);
```

PostgreSQL query để detect long-held connections:

```
-- Connections idle in transaction (leak candidate)
SELECT pid, usename, application_name, state, query_start, state_change,
       EXTRACT(EPOCH FROM (NOW() - state_change)) AS idle_seconds
FROM pg_stat_activity
WHERE state = 'idle in transaction'
  AND state_change < NOW() - INTERVAL '5 minutes'
ORDER BY idle_seconds DESC;

-- Terminate leaked connections
SELECT pg_terminate_backend(pid)
FROM pg_stat_activity
WHERE state = 'idle in transaction'
  AND state_change < NOW() - INTERVAL '30 minutes';
```

Putting It All Together — Architecture Thực Tế [I/A]

Typical production stack

```
Internet
    ↓
[Nginx] (SO_REUSEPORT, HTTP/2, keepalive 200)
    ↓ HTTP/1.1 keepalive upstream
[App Servers] × N (Node/Go/Python)
    ↓                    ↓
[PgBouncer]          [Redis]
(transaction pool,   (connection pool
 pool_size=25)        per-node)
    ↓
[PostgreSQL Primary] → [Replicas]
```

Sizing cheat sheet

```
Layer             Connections     Notes
──────────────────────────────────────────────────────
Users             1,000,000      Concurrent users
Nginx workers     ×65535         Connections per worker
App instances     ×100           Connections per instance to PgBouncer
PgBouncer         ×25            Connections per pool to PostgreSQL
PostgreSQL        max_connections = 200
```

Resultant math:

```
1M users → Nginx (giữ TCP) → App (handle requests concurrently)
App 10 instances × 100 connections = 1000 connections vào PgBouncer
PgBouncer → 25-50 connections thực tế vào PostgreSQL

PostgreSQL RAM:
  50 connections × 10MB = 500MB — hoàn toàn feasible trên 4GB server
```

Config PostgreSQL cho pooling I

```
-- postgresql.conf adjustments khi dùng PgBouncer
max_connections = 200           -- PgBouncer cần ít thôi, set đủ với buffer
superuser_reserved_connections = 3

-- Tăng shared memory (được phân bổ tương ứng max_connections)
shared_buffers = 1GB            -- 25% RAM thường là điểm bắt đầu tốt

-- Statement timeout (tránh query zombie)
statement_timeout = 30000       -- 30 giây

-- Lock timeout
lock_timeout = 10000            -- 10 giây

-- Idle in transaction timeout (phát hiện leak)
idle_in_transaction_session_timeout = 60000  -- 60 giây
```

Monitoring Dashboard — Key Metrics B

```
# PostgreSQL: connections
psql -c "SELECT count(*), state FROM pg_stat_activity GROUP BY state;"

# PgBouncer: pool health
psql -h 127.0.0.1 -p 6432 -U admin pgbouncer -c "SHOW POOLS;"

# Redis: connection info
redis-cli info clients
# connected_clients: N
# blocked_clients: N (BLPOP, BRPOP waiters)
# maxclients: 10000

# System: FD usage
cat /proc/sys/fs/file-nr

# Nginx: active connections
curl http://localhost/nginx_status
# Active connections: 1024

# Application metrics to track:
# - Pool utilization (%) = active / max
# - Wait time for connection (p99 < 10ms is good)
# - Connection errors per minute
# - Pool timeout errors (= pool too small)
```

Quick Reference: Common Mistakes và Fixes

Mistake	Symptom	Fix
Pool size quá lớn	DB CPU 100%, latency tăng	Giảm pool_size, benchmark
Pool size quá nhỏ	`connection timeout` errors	Tăng pool_size hoặc thêm PgBouncer
Không release connection	Pool dần cạn, app hang	`finally: client.release()`
Dùng prepared statements với PgBouncer tx mode	Error "prepared statement does not exist"	Disable prepared statements
Nginx không set HTTP/1.1 upstream	Keepalive không hoạt động	`proxy_http_version 1.1`
SUBSCRIBE dùng pool connection	Timeout các queries khác	Dedicated connection cho pub/sub

Mistake	Symptom	Fix
THP enabled với Redis	Latency spikes ngẫu nhiên	`echo never > .../transparent_hugepage/enabled`
Docker không set ulimit	Container chỉ handle 1024 connections	Set trong daemon.json

19

Caching Strategies — Giảm 99% load bằng multi-layer cache

> Tags: performance, caching, redis, nginx, CDN, scale Levels: B beginner · I intermediate · A advanced

The Math of Caching

Trước khi đi vào kỹ thuật, hãy hiểu tại sao caching lại quan trọng đến vậy.

```
Scenario: 1 triệu users, mỗi user 10 req/phút = 10M req/phút

Không có cache:
  10,000,000 req/phút → DB
  DB chịu tối đa ~50,000 complex queries/phút → CHẾT

Cache hit ratio 95%:
  10,000,000 × 5% miss = 500,000 req/phút → DB
  DB sống sót, server ổn

Cache hit ratio 99%:
  10,000,000 × 1% miss = 100,000 req/phút → DB
  DB thảnh thơi, có headroom để scale

Cache hit ratio 99.9%:
  10,000,000 × 0.1% miss = 10,000 req/phút → DB
  1 DB instance nhỏ handle được 1M concurrent users
```

Key metric: cache hit ratio. Mỗi 1% tăng = giảm 10% load khi đang ở 90%, nhưng ở 99% thì mỗi 0.1% = giảm 50% load. Curve này không tuyến tính.

Công thức tính ROI của caching:

```
DB queries saved = total_requests × hit_ratio
Cost saved = queries_saved × (DB_query_cost - cache_lookup_cost)

Ví dụ:
  DB query avg: 5ms, costs $0.000001 per query
  Redis lookup: 0.1ms, costs $0.00000005 per lookup

  1M req/phút × 99% hit = 990,000 queries saved × $0.000001 = $0.99/phút = ~$1,400/tháng tiết
kiệm
```

Cache Layers (L1 → L2 → L3 → Origin)

Caching tốt nhất là multi-layer. Mỗi layer có trade-off riêng.

```
Request flow:
User → [L1: In-Process] → [L2: Redis] → [L3: CDN/Edge] → [Origin: DB/API]
         0ms latency        0.1-1ms          1-50ms            10-100ms+
         hot data           warm data         static/semi        source of truth
```

L1: In-Process Cache (0 network latency) ℹ

Cache ngay trong process — không cần network round-trip. Nhanh nhất có thể.

Node.js:

```
// node-cache: simple, battle-tested
const NodeCache = require('node-cache');
const cache = new NodeCache({
  stdTTL: 300,          // default TTL 5 phút
  checkperiod: 60,      // cleanup interval
  maxKeys: 10000,       // giới hạn số keys tránh OOM
  useClones: false,     // tắt clone = nhanh hơn nhưng cẩn thận mutation
});

// lru-cache: LRU eviction policy, production grade
const { LRUCache } = require('lru-cache');
const cache = new LRUCache({
  max: 500,             // max 500 items
  maxSize: 50_000_000,  // max 50MB (cần định nghĩa sizeCalculation)
  ttl: 1000 * 60 * 5,  // 5 phút TTL
  sizeCalculation: (value, key) => Buffer.byteLength(JSON.stringify(value)),
  allowStale: true,     // stale-while-revalidate: trả stale trong khi refresh async
  updateAgeOnGet: false,// true = sliding TTL
  fetchMethod: async (key) => await fetchFromDB(key), // auto-fetch on miss
});

// Usage với stale-while-revalidate
const value = await cache.fetch('user:123');
```

Go:

```
// sync.Map: built-in, zero deps, cho data ít thay đổi
var cache sync.Map
cache.Store("key", value)
v, ok := cache.Load("key")

// ristretto: production-grade, admission policy, cost-based eviction
import "github.com/dgraph-io/ristretto"

cache, _ := ristretto.NewCache(&ristretto.Config{
    NumCounters: 1e7,     // số counters để track frequency (10x MaxCost)
    MaxCost:     1 << 30, // max 1GB
    BufferItems: 64,      // per-Get buffer size
    Metrics:     true,    // enable stats
})

// Ristretto dùng TinyLFU admission policy:
// - Track frequency của tất cả keys (kể cả chưa cache)
// - Key mới chỉ vào cache nếu frequency > key hiện tại bị evict
// - Ngăn cache pollution từ one-hit wonders
cache.Set("key", value, cost) // cost = bytes, custom unit
cache.Get("key")
```

Python:

```
# functools.lru_cache: built-in, cho pure functions
from functools import lru_cache

@lru_cache(maxsize=1000)
def get_user_settings(user_id: int) -> dict:
    return db.query(f"SELECT * FROM settings WHERE user_id = {user_id}")

# cachetools: flexible, nhiều eviction policies
from cachetools import TTLCache, LRUCache, cached

ttl_cache = TTLCache(maxsize=10000, ttl=300)  # 5 phút TTL
```

```
@cached(cache=ttl_cache)
def expensive_function(arg):
    return compute(arg)
```

Java — Caffeine (near-optimal):

```
// Caffeine dùng W-TinyLFU: gần tối ưu về hit ratio
LoadingCache<String, User> cache = Caffeine.newBuilder()
    .maximumSize(10_000)
    .expireAfterWrite(Duration.ofMinutes(5))
    .expireAfterAccess(Duration.ofMinutes(2))  // evict nếu không access
    .refreshAfterWrite(Duration.ofMinutes(1))  // async refresh before expire
    .recordStats()                              // enable metrics
    .build(key → userRepository.findById(key)); // auto-load on miss

// refreshAfterWrite: key trick để tránh stampede
// Khi đến refresh time, 1 thread refresh async, các thread khác vẫn nhận stale
```

Sizing guidelines cho L1:

```
Mỗi process: 100MB - 500MB L1 cache thường là sweet spot
Ví dụ với 4 Node.js workers:
  - 4 processes × 200MB = 800MB total L1 cache
  - Tổng RAM server 8GB → còn 7.2GB cho OS, app code, L2 connections

Theo dõi:
  - process.memoryUsage().heapUsed (Node.js)
  - runtime.MemStats.HeapAlloc (Go)
  - jmap -heap PID (Java)
```

Cảnh báo quan trọng về L1: ℹ

```
⚠ Mỗi process có L1 riêng biệt → INCONSISTENCY giữa instances.

    Server A process 1: cache user:123 = {name: "Hieu", plan: "free"}
    Server B process 2: cache user:123 = {name: "Hieu", plan: "pro"}  ← vừa upgrade

    → User thấy plan khác nhau tùy request route đến server nào!

    Giải pháp:
    - Keep TTL ngắn cho mutable data: 30s - 5 phút
    - Chỉ cache immutable data lâu hơn (user settings ít thay đổi)
    - Hoặc dùng cache invalidation broadcast qua Redis pub/sub

⚠ L1 không survive restart → Cold start thundering herd

    Deploy mới → tất cả instances restart → L1 trống → 100% traffic hit L2/DB
    → DB overload ngay sau deploy

    Giải pháp: Cache warming (xem phần cuối)
```

L2: Distributed Cache (Redis/Memcached) ℹ

Cache tập trung, shared giữa tất cả instances. Source of truth cho cached data.

Redis vs Memcached — khi nào dùng gì:

```
Redis:
  ✓ Cần data structures (sorted sets, lists, hashes)
  ✓ Cần persistence (RDB/AOF)
  ✓ Cần Pub/Sub, Streams, Lua scripting
  ✓ Cần atomic operations (INCR, transactions)
  ✓ Cần TTL per key
```

```
Memcached:
  ✓ Pure caching, nothing else
  ✓ Multi-threaded (Redis 6+ cũng single-threaded per shard)
  ✓ Memory efficiency hơn một chút
  ✓ Simpler operational model

→ 90% cases: Redis. Memcached chỉ khi đã profile và thấy Redis là bottleneck.
```

Key Design — quan trọng nhất: I

```
Pattern: {entity}:{id}:{version_or_qualifier}

Ví dụ thực tế:
  user:12345                    ← user object
  user:12345:permissions        ← permissions của user đó
  user:12345:settings:v2        ← version-tagged key
  product:abc-123:price:vn      ← localized price
  session:token_xyz             ← session data
  rate_limit:api:user:12345     ← rate limiting counter
  leaderboard:game:weekly       ← sorted set

Không nên:
  U12345                        ← không rõ entity type
  user_12345_data_v2_new        ← dùng underscore, verbose
  cache:user:12345              ← prefix "cache:" thừa, đã là cache rồi
```

TTL Strategies: I

```python
import redis
import random

r = redis.Redis()

# Strategy 1: Fixed TTL — đơn giản nhất
r.setex("user:123", 300, json.dumps(user_data))  # 5 phút

# Strategy 2: Jittered TTL — tránh thundering herd khi nhiều keys expire cùng lúc
base_ttl = 300
jitter = random.randint(0, 60)  # ±60s ngẫu nhiên
r.setex("user:123", base_ttl + jitter, json.dumps(user_data))

# Strategy 3: Sliding window TTL — extend mỗi khi access
def get_with_sliding_ttl(key, ttl=300):
    pipe = r.pipeline()
    pipe.get(key)
    pipe.expire(key, ttl)  # reset TTL mỗi lần đọc
    results = pipe.execute()
    return results[0]

# Strategy 4: Never expire (background refresh)
# Set TTL dài (24h), dùng background job refresh trước khi expire
r.setex("hot_product:001", 86400, json.dumps(product_data))
# Cron job mỗi 1h check và refresh hot keys
```

Serialization — chọn đúng format: I

```python
import json
import msgpack

data = {"user_id": 12345, "name": "Nguyen Van A", "scores": [1, 2, 3, 4, 5]}

# JSON: readable, universal, ~200 bytes
json_bytes = json.dumps(data).encode()  # 200 bytes

# MessagePack: binary, 30-50% nhỏ hơn JSON, vẫn schema-free
msgpack_bytes = msgpack.packb(data)     # ~120 bytes

# Protobuf: smallest, cần định nghĩa schema trước
```

```
# ~80 bytes nhưng cần .proto file và code generation

# Khi nào dùng gì:
# - Debugging/logging cần readable → JSON
# - High traffic, bandwidth cost → MessagePack
# - Strict schema, cross-language, critical perf → Protobuf
```

Cache Stampede — vấn đề cực kỳ nguy hiểm ở scale: A

```
Scenario:
  - 1 triệu users đang request cùng 1 product page
  - TTL expire đúng lúc flash sale
  - 1 triệu requests miss cache đồng thời
  - 1 triệu DB queries trong vài giây
  → DB crash, site down

Solution 1: Mutex Lock (SETNX)
```

```python
import redis
import time

r = redis.Redis()

def get_with_mutex(key, fetch_func, ttl=300, lock_ttl=10):
    # Try get from cache
    cached = r.get(key)
    if cached:
        return json.loads(cached)

    # Cache miss → try to acquire lock
    lock_key = f"lock:{key}"
    acquired = r.set(lock_key, "1", nx=True, ex=lock_ttl)  # NX = only if not exists

    if acquired:
        try:
            # This process rebuilds the cache
            data = fetch_func()
            r.setex(key, ttl, json.dumps(data))
            return data
        finally:
            r.delete(lock_key)
    else:
        # Another process is rebuilding → wait and retry
        time.sleep(0.1)
        return get_with_mutex(key, fetch_func, ttl, lock_ttl)
        # ! Cần giới hạn retry để tránh infinite loop
```

```python
# Solution 2: Stale-While-Revalidate
def get_stale_while_revalidate(key, fetch_func, ttl=300, stale_ttl=600):
    # Store 2 values: actual data + "fresh" flag
    data = r.get(key)
    is_fresh = r.get(f"{key}:fresh")

    if data and is_fresh:
        return json.loads(data)  # Fresh hit

    if data and not is_fresh:
        # Stale hit → trigger async refresh, return stale immediately
        trigger_background_refresh(key, fetch_func, ttl, stale_ttl)
        return json.loads(data)  # Return stale while revalidating

    # Complete miss → must fetch synchronously
    result = fetch_func()
    r.setex(key, stale_ttl, json.dumps(result))
    r.setex(f"{key}:fresh", ttl, "1")  # fresh flag expires earlier
    return result

# Solution 3: XFetch Algorithm (probabilistic early expiration) A
import math
```

```
def xfetch_get(key, fetch_func, ttl=300, beta=1.0):
    """
    beta: higher = more eager early expiration (1.0 thường là tốt)
    Tự động expire sớm với xác suất tăng dần khi gần đến TTL
    → Nhiều processes sẽ refresh sớm, nhưng phân tán theo thời gian
    """
    result = r.get(key)
    if not result:
        value = fetch_func()
        r.setex(key, ttl, json.dumps({"value": value, "delta": 0, "expires": time.time() + ttl}))
        return value

    cached = json.loads(result)
    remaining_ttl = r.ttl(key)
    delta = cached.get("delta", 1)

    # XFetch: expire early với probability = delta × beta × log(random()) / remaining_ttl
    if -delta * beta * math.log(random.random()) > remaining_ttl:
        # Tự nguyện expire sớm
        start = time.time()
        value = fetch_func()
        delta = time.time() - start  # đo thời gian fetch
        r.setex(key, ttl, json.dumps({"value": value, "delta": delta, "expires": time.time() + tt
l}))
        return value

    return cached["value"]
```

L3: CDN / Edge Cache B I

Cache ở edge gần user nhất. Giảm latency + giảm tải về origin server.

HTTP Cache-Control headers — master guide:

```
# Static assets với content hashing (filename thay đổi khi content thay đổi)
location ~* \.(js|css|png|jpg|gif|svg|woff2)$ {
    # Nếu file có hash trong tên (app.a1b2c3.js) → cache 1 năm
    if ($uri ~* "\.[0-9a-f]{8,}\.(js|css)$") {
        add_header Cache-Control "public, max-age=31536000, immutable";
        # immutable = browser KHÔNG cần revalidate kể cả khi refresh
    }
}

# API responses có thể cache public
location /api/products {
    add_header Cache-Control "public, max-age=60, stale-while-revalidate=300, stale-if-
error=86400";
    # max-age=60: fresh 1 phút
    # stale-while-revalidate=300: serve stale 5 phút trong khi refresh ngầm
    # stale-if-error=86400: serve stale 1 ngày nếu origin error
}

# User-specific → KHÔNG cache public
location /api/user/profile {
    add_header Cache-Control "private, no-store";
    # private = chỉ browser cache, không CDN
    # no-store = không lưu gì cả
}
```

Cloudflare Cache Rules (terraform):

```
resource "cloudflare_ruleset" "cache_rules" {
  zone_id = var.zone_id
  name    = "Cache optimization"
  kind    = "zone"
  phase   = "http_response_headers_transform"
```

```
  rules {
    action = "rewrite"
    action_parameters {
      headers {
        name      = "Cache-Control"
        operation = "set"
        value     = "public, max-age=3600, stale-while-revalidate=86400"
      }
    }
    expression = "(http.request.uri.path matches \"^/api/public/\")"
    enabled    = true
  }
}
```

Surrogate Keys / Cache Tags (Cloudflare/Fastly): A

```python
# Tag cached responses để purge theo group
from flask import Flask, Response
import requests

app = Flask(__name__)

@app.route('/api/products/<product_id>')
def get_product(product_id):
    data = fetch_product(product_id)

    response = Response(json.dumps(data))
    response.headers['Cache-Control'] = 'public, max-age=3600'
    # Tag với product ID và category
    response.headers['Cache-Tag'] = f"product:{product_id},category:{data['category_id']}"
    # Cloudflare dùng Cache-Tag header
    # Fastly dùng Surrogate-Key header
    return response

# Khi update product → purge chỉ các response liên quan
def on_product_update(product_id, category_id):
    # Purge tất cả responses tagged với product này
    requests.post(
        f"https://api.cloudflare.com/client/v4/zones/{ZONE_ID}/purge_cache",
        json={"tags": [f"product:{product_id}"]},
        headers={"Authorization": f"Bearer {CF_TOKEN}"}
    )
```

Lỗi phổ biến với Cache-Control: B

```
! Cache-Control: no-cache KHÔNG CÓ NGHĨA LÀ không cache!
    no-cache = "cache nhưng phải revalidate với server trước khi serve"
    → Browser vẫn lưu, nhưng gửi conditional GET mỗi lần
    → Nếu server trả 304 Not Modified → browser dùng cached version

    Muốn không cache thật sự: Cache-Control: no-store

! Vary: Cookie = cache riêng biệt cho mỗi cookie value
    → 1M users = 1M cache entries riêng biệt
    → CDN cache useless cho authenticated endpoints
    → Chỉ dùng Vary: Accept-Encoding (safe) hoặc Accept-Language (ok)

! Vary: * = "never cache this response at shared caches"
    → CDN hoàn toàn bypass
```

Cache Patterns I A

Cache-Aside (Lazy Loading) — phổ biến nhất

```
def get_user(user_id: int):
    # Step 1: Check cache
    cache_key = f"user:{user_id}"
    cached = redis.get(cache_key)
    if cached:
        return json.loads(cached)

    # Step 2: Cache miss → query DB
    user = db.execute("SELECT * FROM users WHERE id = %s", user_id).fetchone()

    # Step 3: Write to cache
    redis.setex(cache_key, 300, json.dumps(user))

    return user

def update_user(user_id: int, data: dict):
    # Update DB
    db.execute("UPDATE users SET ... WHERE id = %s", user_id)

    # DELETE cache, NOT update
    redis.delete(f"user:{user_id}")
    # ! Tại sao delete không update?

# Race condition nếu update cache:
# T=0: Request A update user → start DB write
# T=1: Request B update user → start DB write
# T=2: Request B finish → write value B to cache
# T=3: Request A finish → write value A to cache (overwrite B!)
# Result: DB has value B (correct), cache has value A (STALE BUG)
# Delete thay vì update → next read sẽ miss và fetch từ DB (correct value)
```

Write-Through I

```
def update_user_write_through(user_id: int, data: dict):
    # Update cache và DB atomically
    pipe = redis.pipeline()
    pipe.setex(f"user:{user_id}", 300, json.dumps(data))
    # Nếu DB write fail sau đây → cache có stale data!
    # Cần transaction hoặc saga pattern cho true atomicity
    pipe.execute()

    db.execute("UPDATE users SET ... WHERE id = %s", user_id)

# Pro: cache luôn có data mới nhất sau write
# Con: write latency tăng (phải write cả cache + DB)
# Dùng khi: data write-once-read-many, consistency quan trọng hơn write speed
```

Write-Behind (Write-Back) A

```
from collections import defaultdict
import threading
import time

class WriteBehindCache:
    def __init__(self, redis_client, db):
        self.redis = redis_client
        self.db = db
        self.dirty_keys = {}
        self.lock = threading.Lock()

        # Background flush thread
        threading.Thread(target=self._flush_loop, daemon=True).start()
```

```
    def set(self, key: str, value: dict):
        # Write to cache immediately
        self.redis.setex(key, 3600, json.dumps(value))

        # Mark as dirty for async DB write
        with self.lock:
            self.dirty_keys[key] = value

    def _flush_loop(self):
        while True:
            time.sleep(1)  # flush every second
            with self.lock:
                dirty = self.dirty_keys.copy()
                self.dirty_keys.clear()

            if dirty:
                # Batch write to DB
                self.db.bulk_upsert(dirty.values())

# ⚠️ Nguy hiểm: nếu cache crash trước khi flush → mất data
# Dùng cho: view counters, click tracking, analytics
# KHÔNG dùng cho: financial data, orders, user credentials

# Ví dụ an toàn hơn: dùng Redis persistence (AOF) để durability
# redis.conf: appendonly yes, appendfsync everysec
```

Nginx Microcaching I A

```
# /etc/nginx/conf.d/microcache.conf

# Định nghĩa cache zone trong memory
proxy_cache_path /var/cache/nginx/microcache
    levels=1:2
    keys_zone=MICROCACHE:10m    # 10MB cho metadata (lưu ~80K keys)
    max_size=1g                  # 1GB disk cho cached content
    inactive=60m                 # xóa content không access trong 60 phút
    use_temp_path=off;           # write thẳng vào cache dir, không qua temp

server {
    location /api/ {
        proxy_cache MICROCACHE;
        proxy_cache_key "$scheme$request_method$host$request_uri";
        proxy_cache_valid 200 1s;          # Cache 200 responses 1 giây
        proxy_cache_valid 404 10s;         # Cache 404 lâu hơn
        proxy_cache_use_stale error timeout updating http_500 http_502 http_503;
        # updating = serve stale khi đang refresh (stale-while-revalidate behavior)

        # Không cache POST hoặc requests có Authorization header
        proxy_cache_bypass $http_authorization $request_method;
        proxy_no_cache $http_authorization;

        # Lock: chỉ 1 request rebuild cache cho 1 key
        proxy_cache_lock on;
        proxy_cache_lock_timeout 5s;

        # Debug header (remove in production)
        add_header X-Cache-Status $upstream_cache_status;

        proxy_pass http://backend;
    }
}
```

Tại sao 1 giây microcache lại mạnh đến vậy:

```
Một endpoint nhận 10,000 req/s:
  - Không cache: 10,000 req/s → backend
  - 1 giây cache: 1 req/s → backend (9,999 từ cache)
  - Cache hit ratio: 99.99%

Với endpoint idempotent (product list, public data):
  1 giây staleness = chấp nhận được
  Giảm 99.99% load = backend có thể chạy trên instance nhỏ hơn 100x
```

HTTP Caching Headers Deep Dive I

ETag — conditional requests:

```python
# Server side: generate ETag từ content hash
import hashlib

@app.route('/api/products/<id>')
def get_product(id):
    product = fetch_product(id)
    etag = hashlib.md5(json.dumps(product, sort_keys=True).encode()).hexdigest()

    # Check if client has current version
    if request.headers.get('If-None-Match') == f'"{etag}"':
        return Response(status=304)  # Not Modified - saves bandwidth!

    response = jsonify(product)
    response.headers['ETag'] = f'"{etag}"'
    response.headers['Cache-Control'] = 'public, max-age=60'
    return response

# Client chỉ download data mới khi thực sự thay đổi
# 304 response: chỉ headers, không body → tiết kiệm bandwidth lớn
```

Cache Partitioning (Chrome 86+): A

```
Chrome partition cache theo: (top-level site, frame site, resource URL)

Ví dụ:
  - site-a.com load jquery.min.js từ cdn.jquery.com
  - site-b.com load jquery.min.js từ cdn.jquery.com

  → Trước Chrome 86: shared cache, cùng một cache entry
  → Sau Chrome 86: 2 cache entries riêng biệt (partitioned by top-level site)

Impact: CDN cross-site caching benefits bị giảm đáng kể.
Solution: Self-host critical third-party resources.
```

Cache Invalidation Strategies A

```
"There are only two hard things in Computer Science:
cache invalidation and naming things."
— Phil Karlton
```

So sánh các strategies:

```
Strategy          | Consistency | Complexity | Latency Impact
------------------|-------------|------------|----------------
TTL-based         | Eventual    | Low        | None
Event-based       | Strong      | Medium     | Slight (pub/sub)
Version-based     | Strong      | Medium     | None
```

```
Tag-based          | Strong      | High       | None
CDC (Debezium)     | Near-real   | High       | None
```

Event-based invalidation với Redis Pub/Sub:

```python
# Publisher (khi data thay đổi)
def update_product(product_id: int, data: dict):
    db.update_product(product_id, data)

    # Publish invalidation event
    redis.publish(
        channel="cache:invalidate",
        message=json.dumps({
            "entity": "product",
            "id": product_id,
            "action": "update"
        })
    )

# Subscriber (chạy trên mỗi server instance)
def cache_invalidation_worker():
    pubsub = redis.pubsub()
    pubsub.subscribe("cache:invalidate")

    for message in pubsub.listen():
        if message["type"] == "message":
            event = json.loads(message["data"])

            # Invalidate L1 cache on this instance
            local_cache.delete(f"{event['entity']}:{event['id']}")

            # L2 cache cũng xóa (nếu cần)
            redis.delete(f"{event['entity']}:{event['id']}")
```

Version-based invalidation (safe, no coordination needed): I

```python
# Key bao gồm version → version bump = old keys tự expired hoặc orphaned
def get_versioned_cache_key(entity: str, id: int) -> str:
    # Version lưu trong Redis hoặc config
    version = redis.get(f"cache_version:{entity}") or "v1"
    return f"{entity}:{id}:{version.decode()}"

def invalidate_entity(entity: str):
    # Bump version → tất cả old keys tự nhiên trở nên orphaned
    # Chúng sẽ expire theo TTL, không cần xóa thủ công
    current_version = int(redis.get(f"cache_version:{entity}") or 1)
    redis.set(f"cache_version:{entity}", f"v{current_version + 1}")

# Ví dụ:
# product:123:v1 → invalidate → product:123:v2 (v1 key còn đó nhưng không ai query)
# v1 key expire theo TTL → memory tự dọn
```

Cache Warming I A

```python
# Phân tích access patterns để biết hot keys
def analyze_hot_keys():
    """Dùng Redis MONITOR hoặc keyspace notifications để track"""
    # Redis command:
    # redis-cli --hotkeys -i 0.01  # sample 1% keys để tìm hot keys
    # redis-cli OBJECT FREQ key     # xem frequency (cần maxmemory-policy allkeys-lfu)
    pass

# Pre-warm cache sau deploy
async def warm_cache_on_startup():
    # Top 1000 products by traffic (từ analytics)
```

```
    hot_products = await analytics_db.query("""
        SELECT product_id, COUNT(*) as views
        FROM page_views
        WHERE created_at > NOW() - INTERVAL '7 days'
        GROUP BY product_id
        ORDER BY views DESC
        LIMIT 1000
    """)

    # Pre-load với concurrency để warm nhanh
    semaphore = asyncio.Semaphore(50)  # max 50 concurrent DB queries

    async def warm_one(product_id):
        async with semaphore:
            product = await db.get_product(product_id)
            await redis.setex(
                f"product:{product_id}",
                300 + random.randint(0, 60),  # jittered TTL
                json.dumps(product)
            )

    await asyncio.gather(*[warm_one(p.product_id) for p in hot_products])
    logger.info(f"Cache warmed: {len(hot_products)} hot products")

# Canary deployment để warm gradually:
# 1. Deploy new version đến 1% instances
# 2. 1% traffic warms cache trên instances đó
# 3. Gradually increase traffic đến 100%
# 4. Khi rollout 100%, tất cả instances đã warm
```

Monitoring Cache Health ℹ

```
# Redis metrics quan trọng nhất
redis-cli INFO stats | grep -E "keyspace_hits|keyspace_misses|evicted_keys|used_memory"

# Tính hit ratio từ Redis stats
redis-cli INFO stats | awk '
/keyspace_hits/ { hits = $2 }
/keyspace_misses/ { misses = $2 }
END { printf "Hit ratio: %.2f%%\n", hits/(hits+misses)*100 }'

# Memory usage
redis-cli INFO memory | grep -E "used_memory_human|maxmemory_human|mem_fragmentation_ratio"
# mem_fragmentation_ratio > 1.5 = Redis cần restart để defrag
# mem_fragmentation_ratio < 1.0 = Redis đang swap ra disk (nguy hiểm!)

# Slow log (queries > 10ms)
redis-cli SLOWLOG GET 10

# Monitor evictions (nếu evicted_keys tăng = cache quá nhỏ)
redis-cli INFO stats | grep evicted_keys
```

Prometheus metrics để alert:

```
# Alert khi cache hit ratio < 80%
- alert: LowCacheHitRatio
  expr: |
    redis_keyspace_hits_total /
    (redis_keyspace_hits_total + redis_keyspace_misses_total) < 0.80
  for: 5m
  annotations:
    summary: "Cache hit ratio thấp: {{ $value | humanizePercentage }}"

# Alert khi memory > 90%
- alert: RedisMemoryHigh
```

```
  expr: redis_memory_used_bytes / redis_memory_max_bytes > 0.90
  for: 2m
```

Quick Reference — Chọn Strategy cho từng use case

```
Use case                          → Strategy                    TTL
----------------------------------+-----------------------------+----------
User profile                      → Cache-aside + delete        5 phút
Product catalog (public)          → Write-through + CDN         1 giờ
Inventory count                   → Write-behind + Redis INCR   N/A (realtime)
Search results                    → Cache-aside + jitter        10 phút
Session data                      → Write-through               30 phút (sliding)
API responses (public)            → Nginx microcache            1-60 giây
Static assets (hashed)            → CDN + immutable             1 năm
Static assets (non-hashed)        → CDN + short max-age         1 ngày
Leaderboard                       → Redis sorted set + TTL      1 phút
Rate limit counters               → Redis INCR + EXPIRE         window period
Feature flags                     → L1 (in-process) + TTL      30 giây
```

Checklist cho Production Caching

```
☐ Đo cache hit ratio trước khi và sau khi implement
☐ Jitter TTLs để tránh stampede
☐ Implement circuit breaker: nếu cache down, gracefully fallback to DB
☐ Không cache sensitive data (passwords, tokens) trừ khi encrypted
☐ Cache key design: entity:id:qualifier pattern
☐ Monitoring: hit ratio, memory usage, eviction rate, latency
☐ Cache warming strategy cho cold start
☐ Test cache invalidation trong integration tests
☐ Document TTL decisions và rationale
☐ Có kế hoạch xử lý khi Redis down (fallback, not crash)
```

20

Database Optimization at Scale — PostgreSQL/MySQL chịu 100K queries/sec

Tags: postgresql, mysql, database, performance, indexing, partitioning, scale Levels: B beginner · I intermediate · A advanced

The Math of Database Performance

```
PostgreSQL trên typical 8-core, 32GB RAM server:
  Simple reads (index scan):        ~50,000 - 100,000 queries/sec
  Complex joins (multi-table):      ~5,000 - 20,000 queries/sec
  Bulk writes (batched):            ~10,000 - 50,000 rows/sec
  Full-table seq scan (1M rows):    ~500ms - 2s

Với optimization:
  Covering index → Index-only scan: 10x faster (no heap lookup)
  Partial index → filter 99%:       100x less I/O
  Batch insert vs individual:       50-100x faster write throughput
  PgBouncer transaction pooling:    10x more concurrent connections
  Read replica:                     2-10x read throughput (horizontal)
  Partitioning (hot/cold):          10-100x faster queries on recent data
```

Connection Management B I

Tại sao connections là vấn đề

```
PostgreSQL tạo 1 OS process per connection (không phải thread).
1000 connections = 1000 OS processes = ~800MB RAM chỉ cho process overhead.

max_connections = 1000:
  Mỗi backend process: ~5-10MB RAM
  1000 connections × 8MB = 8GB RAM chỉ cho idle connections
  + work_mem per query = OOM territory

max_connections = 200:
  200 × 8MB = 1.6GB → manageable
  Dùng PgBouncer để multiplex 1000 app connections → 200 DB connections
```

PgBouncer Transaction Pooling

```
# /etc/pgbouncer/pgbouncer.ini

[databases]
mydb = host=localhost port=5432 dbname=mydb

[pgbouncer]
```

```
listen_port = 6432
listen_addr = *

pool_mode = transaction          # session | transaction | statement
# transaction: connection trả về pool sau mỗi transaction → RECOMMENDED
# session: connection giữ suốt session → pooling ít hiệu quả
# statement: trả về sau mỗi statement → không dùng được với multi-statement tx

max_client_conn = 1000           # app có thể mở 1000 connections đến pgbouncer
default_pool_size = 50           # chỉ 50 connections thật đến PostgreSQL
min_pool_size = 10               # luôn giữ ít nhất 10 connections sẵn sàng
reserve_pool_size = 10           # emergency pool khi tải đột biến
reserve_pool_timeout = 3         # chờ 3s trước khi dùng reserve pool

# Authentication
auth_type = scram-sha-256
auth_file = /etc/pgbouncer/userlist.txt

# Timeouts
query_timeout = 30               # kill query sau 30s
client_idle_timeout = 600        # close idle client sau 10 phút
server_idle_timeout = 600        # close idle server connection sau 10 phút
connect_timeout = 5              # timeout khi connect đến PostgreSQL

# Stats
stats_period = 60                # log stats mỗi 60s
```

```
# Monitor pgbouncer
psql -p 6432 -U pgbouncer pgbouncer -c "SHOW POOLS;"
# Xem: cl_active, cl_waiting, sv_active, sv_idle, sv_used

# Alerts khi cl_waiting > 0 sustained → pool quá nhỏ
# Tăng default_pool_size hoặc tăng max_connections của PostgreSQL
```

Rule of thumb cho connection sizing: I

```
PostgreSQL max_connections = (CPU_cores × 2) + effective_spindles
  Với 8 cores, SSD (effective_spindles ≈ 1):
  max_connections = 8 × 2 + 1 = 17 → round up to practical: 100-200

  Lý do:
  - Mỗi core xử lý 1 query tại 1 thời điểm
  - I/O bound queries: một số connections chờ I/O, CPU còn cores khác làm việc
  - Quá nhiều hơn = context switching overhead outweigh benefit

PgBouncer pool per DB per user:
  default_pool_size = max_connections / số_databases_trong_pgbouncer
  Ví dụ: max_connections = 100, 3 databases → pool_size ≈ 30 mỗi DB
```

Query Optimization I A

EXPLAIN ANALYZE — đọc execution plan

```
-- Basic explain
EXPLAIN SELECT * FROM orders WHERE user_id = 123;

-- Full analysis với actual timing và buffer stats (QUAN TRỌNG nhất)
EXPLAIN (ANALYZE, BUFFERS, FORMAT TEXT, VERBOSE)
  SELECT o.*, u.name, u.email
  FROM orders o
  JOIN users u ON u.id = o.user_id
  WHERE o.created_at > NOW() - INTERVAL '7 days'
    AND o.status = 'pending'
  ORDER BY o.created_at DESC
```

```
  LIMIT 100;

-- Output có thể paste vào https://explain.depesz.com hoặc https://explain.dalibo.com
-- để visualize dạng tree
```

Đọc output EXPLAIN: I

```
Hash Join  (cost=250.00..5420.50 rows=1200 width=180)
           (actual time=45.234..892.445 rows=987 loops=1)
   Buffers: shared hit=2341 read=890 written=12
  →  Seq Scan on orders  (cost=0.00..4890.00 rows=15000 width=120)
          (actual time=0.023..750.123 rows=15000 loops=1)
        Filter: (status = 'pending')
        Rows Removed by Filter: 285000
        Buffers: shared hit=1890 read=890

Đọc từ dưới lên (leaf nodes trước):
  cost=0.00..4890.00 → startup cost .. total cost (arbitrary units)
  rows=15000 → estimated rows (so với actual rows để thấy estimate accuracy)
  actual time=0.023..750.123 → actual ms (startup..total)
  Buffers shared hit=1890 → từ PG shared_buffers (RAM) → FAST
  Buffers shared read=890  → từ disk/OS cache → SLOW
  Rows Removed by Filter: 285000 → đọc 300K rows, bỏ 285K → cần index!
```

Các node types và ý nghĩa: I

```
Seq Scan:
  - Đọc toàn bộ table
  - OK cho tables nhỏ (<8MB) hoặc khi lấy >10% rows
  - Vấn đề khi lấy <1% rows từ table lớn

Index Scan:
  - Dùng index để find rows, sau đó fetch từ heap (table)
  - Good khi selectivity cao (lấy ít rows)
  - Mỗi row cần 2 lookups: index + heap

Index Only Scan:
  - Tất cả data cần có trong index (covering index)
  - Không cần fetch heap → 2-10x faster hơn Index Scan
  - Visibility map phải up-to-date (cần regular VACUUM)

Bitmap Index Scan + Bitmap Heap Scan:
  - PG build bitmap of matching pages, rồi fetch pages theo thứ tự
  - Tốt hơn Index Scan cho medium selectivity (1-10% rows)
  - Giảm random I/O bằng cách sort page access

Hash Join:
  - Build hash table từ smaller table, probe với larger
  - Tốt cho equijoin trên large datasets
  - Cần work_mem để hold hash table

Nested Loop:
  - Outer loop qua rows, inner loop find matches via index
  - Tốt khi outer set nhỏ và inner có index
  - Tệ khi outer set lớn (O(n×m) complexity)

Merge Join:
  - Sort cả hai sides, merge sorted streams
  - Tốt khi cả hai sides đã sorted (index scan)
```

Khi PostgreSQL chọn Seq Scan dù có index: I

```
-- PG dùng statistics để estimate selectivity
-- Nếu estimate sai → wrong plan

-- Xem statistics
SELECT schemaname, tablename, attname, n_distinct, correlation
FROM pg_stats
```

```
WHERE tablename = 'orders' AND attname = 'status';

-- n_distinct = số unique values
-- correlation = physical ordering correlation với logical ordering
--   1.0 = perfectly sorted (BRIN index hiệu quả)
--   0.0 = random order

-- Update statistics sau bulk load
ANALYZE orders;
-- Hoặc specific columns
ANALYZE orders(status, created_at);

-- Thử force index để test (KHÔNG dùng production)
SET enable_seqscan = off;
EXPLAIN SELECT * FROM orders WHERE status = 'pending';
SET enable_seqscan = on;

-- Nếu sau khi force index, actual time vẫn chậm hơn seq scan
-- → PG đúng, không cần index cho query này
```

Indexing Strategies I A

Index types và khi nào dùng

```
-- B-tree (default): equality, range, sort, LIKE 'prefix%'
CREATE INDEX idx_orders_user_id ON orders(user_id);
CREATE INDEX idx_orders_created_at ON orders(created_at);

-- Composite: thứ tự cột quan trọng!
-- Rule: equality columns trước, range columns sau
CREATE INDEX idx_orders_user_status_date ON orders(user_id, status, created_at);
-- Hiệu quả cho:   WHERE user_id = X AND status = 'pending' AND created_at > Y
-- Không hiệu quả: WHERE status = 'pending' AND created_at > Y (không có user_id)

-- Hash: chỉ equality, không support range/sort (ít dùng)
CREATE INDEX idx_sessions_token ON sessions USING HASH (token);
-- Nhỏ hơn B-tree, faster cho pure equality lookup
```

Partial Index — index thông minh nhất: I

```
-- Chỉ index rows bạn thực sự query
-- Bài toán: 10M orders, chỉ 50K là 'pending', queries chủ yếu filter pending

-- Index toàn bộ:
CREATE INDEX idx_orders_status ON orders(status);
-- Index size: 10M rows × ~30 bytes = ~300MB

-- Partial index:
CREATE INDEX idx_orders_pending ON orders(created_at)
WHERE status = 'pending';
-- Index size: 50K rows × ~30 bytes = ~1.5MB (200x smaller!)
-- Faster updates (chỉ update index khi status = 'pending')

-- Thêm ví dụ:
CREATE INDEX idx_users_unverified ON users(created_at)
WHERE email_verified = false;
-- Chỉ index users chưa verify email (thường rất nhỏ so với total users)

CREATE INDEX idx_jobs_queued ON background_jobs(priority DESC, created_at ASC)
WHERE status = 'queued';
-- Worker SELECT ... ORDER BY priority DESC, created_at ASC WHERE status = 'queued'
-- Index-only scan trên subset nhỏ → cực nhanh
```

Covering Index (INCLUDE) — xóa heap lookup: I A

```
-- Vấn đề: Index Scan cần 2 lookups (index + heap) cho mỗi row
-- Solution: INCLUDE thêm columns vào index để avoid heap lookup

-- Query: SELECT name, email FROM users WHERE user_id = 12345
-- Thông thường: B-tree index trên user_id, sau đó fetch heap cho name, email

-- Covering index: INCLUDE name và email trong index
CREATE INDEX idx_users_id_covering ON users(user_id) INCLUDE (name, email);
-- Kết quả: Index-Only Scan, không cần fetch heap
-- Khi nào dùng: high-frequency queries lấy specific columns từ PK lookup

-- INCLUDE vs composite:
CREATE INDEX idx1 ON users(user_id, name, email);  -- có thể dùng index cho WHERE name = X
CREATE INDEX idx2 ON users(user_id) INCLUDE (name, email);  -- không WHERE name, chỉ SELECT
-- idx2 nhỏ hơn vì INCLUDE columns không indexable, chỉ stored
```

Expression Index: I

```
-- Query thường dùng lower() cho case-insensitive search
SELECT * FROM users WHERE lower(email) = lower('User@Example.com');

-- Không có expression index: seq scan hoặc index scan + evaluate lower() per row
-- Với expression index:
CREATE INDEX idx_users_email_lower ON users(lower(email));
-- PG lưu lower(email) trong index → query có thể dùng trực tiếp

-- Ví dụ khác:
CREATE INDEX idx_events_date ON events(DATE(created_at));
-- WHERE DATE(created_at) = '2026-01-15' → sử dụng index

CREATE INDEX idx_products_search ON products(to_tsvector('english', name || ' ' || description));
-- Full-text search index

-- ! Expression index: PG phải evaluate expression khi INSERT/UPDATE
-- → Chỉ tạo khi query pattern đủ thường xuyên để justify overhead
```

BRIN Index — cho time-series data: I A

```
-- BRIN = Block Range Index
-- Lưu min/max per block range (128 pages default)
-- Cực nhỏ, cực nhanh cho append-only / naturally ordered data

-- Bảng logs 1 tỷ rows, tăng theo thời gian:
CREATE INDEX idx_logs_created_brin ON logs USING BRIN(created_at)
WITH (pages_per_range = 128);

-- Index size comparison:
-- B-tree: 1B rows × 30 bytes = ~30GB
-- BRIN:   1B rows / 128 pages × 16 bytes = ~10MB (3000x smaller!)

-- Hiệu quả khi: WHERE created_at BETWEEN '2026-01-01' AND '2026-01-31'
-- PG check min/max của mỗi block range → skip ranges không match
-- Correlation cao (data physically ordered theo time) → rất hiệu quả
-- Correlation thấp (random inserts) → không hiệu quả

-- Kiểm tra correlation:
SELECT correlation FROM pg_stats
WHERE tablename = 'logs' AND attname = 'created_at';
-- > 0.9: BRIN excellent
-- < 0.5: B-tree tốt hơn
```

GIN Index — cho JSONB, arrays, full-text: I

```
-- JSONB: query nested data
CREATE INDEX idx_products_attributes ON products USING GIN(attributes jsonb_path_ops);
-- attributes = {"color": "red", "size": "XL", "tags": ["sale", "new"]}
-- WHERE attributes @> '{"color": "red"}' → sử dụng GIN index
```

```
-- Array containment
CREATE INDEX idx_posts_tags ON posts USING GIN(tags);
-- WHERE tags @> ARRAY['postgresql', 'performance']

-- Full-text search
CREATE INDEX idx_articles_fts ON articles USING GIN(to_tsvector('english', content));
-- WHERE to_tsvector('english', content) @@ plainto_tsquery('database optimization')
```

Audit unused indexes — thường xuyên: I

```
-- Indexes không bao giờ được dùng = write overhead lãng phí
SELECT
    schemaname,
    tablename,
    indexname,
    idx_scan,
    idx_tup_read,
    idx_tup_fetch,
    pg_size_pretty(pg_relation_size(indexrelid)) AS index_size
FROM pg_stat_user_indexes
WHERE idx_scan = 0
  AND schemaname NOT IN ('pg_catalog', 'pg_toast')
ORDER BY pg_relation_size(indexrelid) DESC;

-- Indexes ít được dùng (có thể xem xét xóa)
SELECT
    schemaname || '.' || tablename AS table,
    indexname,
    idx_scan,
    pg_size_pretty(pg_relation_size(indexrelid)) AS size,
    pg_size_pretty(pg_total_relation_size(indexrelid)) AS total_size
FROM pg_stat_user_indexes
WHERE idx_scan < 100  -- dưới 100 lần scan kể từ lần cuối pg_stat reset
ORDER BY pg_relation_size(indexrelid) DESC
LIMIT 20;

-- ! Reset stats sau major data changes để có baseline chính xác
SELECT pg_stat_reset();
-- Run workload 1-2 tuần → re-check
```

Write amplification với nhiều indexes: A

```
1 INSERT vào table có 10 indexes:
  1 write to heap (table)
  + 10 writes to index pages
  + WAL entries cho tất cả 11 writes
  = 11x+ write amplification

1 UPDATE (worst case, tất cả indexed columns thay đổi):
  1 write new heap tuple
  1 mark old heap tuple deleted
  + 10 index updates (delete old entry + insert new)
  = 22x amplification

→ Audit indexes trước khi add. Mỗi index cần justify read benefit vs write cost.
→ Trên write-heavy tables: ít hơn 3-5 indexes là ideal
```

Read Replicas Pattern I

```
Topology điển hình:

App Servers
    │
    ├── PgBouncer Primary ──→ PostgreSQL Primary (writes + critical reads)
    │                                    │
```

```
    │                                  Streaming Replication
    │                                          │
    └── PgBouncer Replica ──→ PostgreSQL Replica × 2 (read-heavy queries)
```

```python
# Application-level read/write splitting
import psycopg2

PRIMARY_DSN = "postgresql://user:pass@primary:5432/db"
REPLICA_DSN = "postgresql://user:pass@replica:5432/db"

def get_db(for_write=False):
    """Route to primary for writes, replica for reads"""
    dsn = PRIMARY_DSN if for_write else REPLICA_DSN
    return psycopg2.connect(dsn)

# Read-your-writes consistency problem:
# User update profile → written to primary
# User refresh page → read from replica → thấy OLD data (replication lag!)

# Solution: sticky routing sau write
import time
from functools import wraps

class DatabaseRouter:
    def __init__(self):
        self._use_primary_until = {}  # user_id → timestamp
        self.STICKY_DURATION = 5  # giây

    def mark_wrote(self, user_id: int):
        self._use_primary_until[user_id] = time.time() + self.STICKY_DURATION

    def get_conn(self, user_id: int, for_write=False):
        if for_write:
            self.mark_wrote(user_id)
            return psycopg2.connect(PRIMARY_DSN)

        # Sau khi write, route đến primary trong STICKY_DURATION giây
        until = self._use_primary_until.get(user_id, 0)
        if time.time() < until:
            return psycopg2.connect(PRIMARY_DSN)

        return psycopg2.connect(REPLICA_DSN)
```

Monitoring replication lag:

```sql
-- Trên Primary: xem trạng thái replication
SELECT
    client_addr,
    state,
    sent_lsn,
    write_lsn,
    flush_lsn,
    replay_lsn,
    (sent_lsn - replay_lsn) AS replication_lag_bytes,
    write_lag,
    flush_lag,
    replay_lag
FROM pg_stat_replication;

-- Trên Replica: xem lag
SELECT now() - pg_last_xact_replay_timestamp() AS replication_delay;
-- < 1s: excellent
-- 1-10s: acceptable
-- > 10s: investigate (network, slow queries on replica, I/O bottleneck)

-- Alert khi lag > 30 giây
```

Partitioning I A

Range Partitioning theo date — pattern phổ biến nhất

```
-- Tạo partitioned table
CREATE TABLE events (
    id          BIGSERIAL,
    user_id     BIGINT NOT NULL,
    event_type  VARCHAR(50) NOT NULL,
    properties  JSONB,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
) PARTITION BY RANGE (created_at);

-- Tạo partitions theo tháng
CREATE TABLE events_2026_01 PARTITION OF events
    FOR VALUES FROM ('2026-01-01') TO ('2026-02-01');

CREATE TABLE events_2026_02 PARTITION OF events
    FOR VALUES FROM ('2026-02-01') TO ('2026-03-01');

-- Index trên mỗi partition (tạo riêng hoặc sẽ inherit từ parent)
CREATE INDEX ON events_2026_01(user_id, created_at);
CREATE INDEX ON events_2026_02(user_id, created_at);

-- Tự động tạo partition hàng tháng (pg_partman extension)
SELECT partman.create_parent(
    p_parent_table := 'public.events',
    p_control := 'created_at',
    p_type := 'range',
    p_interval := '1 month',
    p_premake := 3  -- tạo sẵn 3 tháng tới
);
```

Partition pruning — tại sao partitioning giúp performance: I

```
-- Query chỉ lấy dữ liệu tháng này
EXPLAIN SELECT COUNT(*) FROM events
WHERE created_at ⩾ '2026-05-01' AND created_at < '2026-06-01';

-- Kết quả:
-- Aggregate (cost=...)
--   → Seq Scan on events_2026_05 (cost=...)  ← CHỈ scan partition tháng 5!
--
-- Nếu có 24 partitions (2 năm), PG skip 23 partitions
-- events_2026_05: 5M rows
-- events tổng: 120M rows
-- Query chỉ read 5M / 120M = 4.2% data
-- Performance improvement: ~24x ít I/O

-- ! Partition key phải xuất hiện trong WHERE clause để pruning work
-- WHERE user_id = 123 → KHÔNG pruned (user_id không phải partition key)
-- WHERE created_at > X → pruned ✓
```

Drop old partitions — fast delete: A

```
-- Xóa dữ liệu cũ hơn 1 năm
-- Cách thông thường: DELETE FROM events WHERE created_at < NOW() - INTERVAL '1 year'
-- → Lock table, slow, tạo bloat

-- Với partitioning: drop partition = instant!
DROP TABLE events_2025_01;  -- drop toàn bộ partition trong milliseconds!

-- Hoặc detach rồi archive
ALTER TABLE events DETACH PARTITION events_2025_01;
-- events_2025_01 vẫn tồn tại như standalone table
-- Có thể dump ra file, rồi drop
```

```
COPY events_2025_01 TO '/archive/events_2025_01.csv' CSV;
DROP TABLE events_2025_01;
```

Constraint exclusion và cảnh báo: A

```
-- ❗ Unique constraints phải bao gồm partition key
-- Không thể: UNIQUE(id) trên partitioned table
-- Phải: UNIQUE(id, created_at) vì cross-partition unique check không supported

-- ❗ Foreign key FROM partitioned table sang khác: OK
-- ❗ Foreign key TO partitioned table: KHÔNG supported

-- ❗ Cross-partition queries có thể chậm hơn nếu không pruned
SELECT * FROM events WHERE user_id = 123;  -- scan TẤT CẢ partitions
-- Solution: include created_at trong query khi có thể
SELECT * FROM events WHERE user_id = 123 AND created_at > NOW() - INTERVAL '30 days';
```

Materialized Views I A

```
-- Vấn đề: dashboard query phức tạp mất 30s
-- Query nguyên bản:
SELECT
    DATE_TRUNC('day', o.created_at) AS order_date,
    p.category_id,
    COUNT(*) AS order_count,
    SUM(o.total_amount) AS revenue,
    AVG(o.total_amount) AS avg_order_value,
    COUNT(DISTINCT o.user_id) AS unique_customers
FROM orders o
JOIN order_items oi ON oi.order_id = o.id
JOIN products p ON p.id = oi.product_id
WHERE o.created_at ⩾ NOW() - INTERVAL '90 days'
GROUP BY 1, 2
ORDER BY 1 DESC, 3 DESC;
-- Execution time: 45 giây trên 50M orders

-- Materialized view: pre-compute và store kết quả
CREATE MATERIALIZED VIEW mv_daily_revenue AS
    [query trên]
WITH DATA;  -- populate ngay khi tạo

-- Tạo index trên mat view
CREATE UNIQUE INDEX ON mv_daily_revenue(order_date, category_id);

-- Query mat view: ms thay vì giây
SELECT * FROM mv_daily_revenue
WHERE order_date ⩾ NOW() - INTERVAL '30 days';
-- Execution time: 5ms

-- Refresh định kỳ
REFRESH MATERIALIZED VIEW CONCURRENTLY mv_daily_revenue;
-- CONCURRENTLY: không lock, readers vẫn thấy old data trong khi refresh
-- Cần UNIQUE index để CONCURRENTLY hoạt động
-- Không có CONCURRENTLY → lock suốt quá trình refresh (nguy hiểm production)
```

Refresh strategies:

```
-- Strategy 1: Cron job
-- crontab: 0 * * * * psql -c "REFRESH MATERIALIZED VIEW CONCURRENTLY mv_daily_revenue"

-- Strategy 2: Trigger on source table change (near real-time)
CREATE OR REPLACE FUNCTION refresh_mv_daily_revenue()
RETURNS TRIGGER AS $$
BEGIN
```

```
    REFRESH MATERIALIZED VIEW CONCURRENTLY mv_daily_revenue;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER tg_refresh_mv_on_order_change
AFTER INSERT OR UPDATE OR DELETE ON orders
FOR EACH STATEMENT EXECUTE FUNCTION refresh_mv_daily_revenue();
-- ⚠ Cẩn thận với strategy này: mỗi write trigger refresh → overhead lớn
-- Tốt hơn: debounce, chỉ refresh nếu last refresh > 1 phút trước

-- Strategy 3: On-demand với staleness check
SELECT
    schemaname,
    matviewname,
    last_refresh
FROM pg_matviews
WHERE matviewname = 'mv_daily_revenue';
-- Nếu last_refresh > 1 giờ → trigger refresh trong background
```

SKIP LOCKED Pattern — Queue trong Database I A

```
-- Background job queue không cần RabbitMQ/SQS cho < 10K jobs/sec

CREATE TABLE background_jobs (
    id          BIGSERIAL PRIMARY KEY,
    type        VARCHAR(50) NOT NULL,
    payload     JSONB NOT NULL,
    status      VARCHAR(20) NOT NULL DEFAULT 'queued',
    priority    INT NOT NULL DEFAULT 5,
    attempts    INT NOT NULL DEFAULT 0,
    max_retries INT NOT NULL DEFAULT 3,
    run_after   TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX idx_jobs_queue ON background_jobs(priority DESC, created_at ASC)
WHERE status = 'queued';  -- partial index!

-- Worker query: grab 1 job, skip locked rows (other workers)
BEGIN;

WITH next_job AS (
    SELECT id FROM background_jobs
    WHERE status = 'queued'
      AND run_after ⩽ NOW()
      AND attempts < max_retries
    ORDER BY priority DESC, created_at ASC
    LIMIT 1
    FOR UPDATE SKIP LOCKED  -- KEY: bỏ qua rows đang bị lock bởi worker khác
)
UPDATE background_jobs SET
    status = 'processing',
    attempts = attempts + 1,
    updated_at = NOW()
WHERE id = (SELECT id FROM next_job)
RETURNING *;

COMMIT;

-- Với SKIP LOCKED:
-- Worker 1 locks job #1 → Worker 2 skip job #1, lấy job #2
-- Không có deadlock, không có contention
-- 100 workers poll đồng thời = OK, mỗi worker lấy job riêng

-- Complete job
```

```
UPDATE background_jobs
SET status = 'completed', updated_at = NOW()
WHERE id = $1;

-- Fail job (retry)
UPDATE background_jobs
SET status = CASE WHEN attempts >= max_retries THEN 'failed' ELSE 'queued' END,
    run_after = NOW() + (attempts * INTERVAL '1 minute'),  -- exponential backoff
    updated_at = NOW()
WHERE id = $1;
```

Throughput expectations:

```
SKIP LOCKED throughput với dedicated table:
  1 worker:   ~1,000 jobs/sec
  10 workers: ~5,000-8,000 jobs/sec (contention overhead)
  100 workers: ~10,000-20,000 jobs/sec (diminishing returns)

Vượt quá 10,000 jobs/sec: consider Redis-based queue (BullMQ, Sidekiq)
Dưới 10,000 jobs/sec: PostgreSQL SKIP LOCKED là đủ và đơn giản hơn nhiều
```

Batch Operations B I

Batch INSERT — 100x faster

```
import psycopg2
from psycopg2.extras import execute_values, execute_batch

conn = psycopg2.connect(DSN)
cur = conn.cursor()

# Tệ: individual inserts
for user in users:
    cur.execute("INSERT INTO users (name, email) VALUES (%s, %s)",
                (user['name'], user['email']))
# 10,000 inserts = 10,000 round-trips = slow

# Tốt: batch insert
execute_values(
    cur,
    "INSERT INTO users (name, email) VALUES %s",
    [(u['name'], u['email']) for u in users],
    page_size=1000  # batch size per statement
)
# 10,000 inserts = 10 round-trips = ~100x faster

# Nhanh nhất: COPY FROM STDIN
import io

buffer = io.StringIO()
for user in users:
    buffer.write(f"{user['name']}\t{user['email']}\n")
buffer.seek(0)

cur.copy_from(buffer, 'users', columns=('name', 'email'))
# COPY bypasses SQL parser, query planner → fastest possible bulk insert
# 1M rows trong vài giây vs vài phút với individual inserts
```

COPY performance math:

```
1M rows, each row ~100 bytes:
  Individual INSERT: 1M × (network RTT 0.5ms + parse 0.1ms + plan 0.1ms) = ~11 phút
```

```
  Batch INSERT (1000/batch): 1000 × 1ms = 1 giây
  COPY FROM STDIN: ~3-5 giây (I/O bound only, no parse/plan overhead)
```

Batch UPDATE — tránh N+1: I

```
-- Tệ: N+1 pattern
-- Python: for user in users: db.execute("UPDATE users SET score = %s WHERE id = %s")

-- Tốt: batch update với VALUES
UPDATE users AS u
SET score = v.score,
    rank = v.rank
FROM (VALUES
    (1, 100, 1),
    (2, 95, 2),
    (3, 88, 3)
) AS v(id, score, rank)
WHERE u.id = v.id;

-- Với psycopg2:
execute_values(
    cur,
    """UPDATE users AS u SET score = v.score, rank = v.rank
       FROM (VALUES %s) AS v(id, score, rank)
       WHERE u.id = v.id""",
    [(user.id, user.score, user.rank) for user in users],
    template="(%s, %s, %s)"
)
```

Large DELETE — tránh lock toàn bộ table: I

```
-- Xóa 50M rows cũ: ĐỪNG chạy 1 lần
DELETE FROM logs WHERE created_at < '2025-01-01';
-- → Lock rows suốt quá trình → block reads/writes → site down

-- Thay vào đó: chunk deletes
DO $$
DECLARE
    deleted_count INT;
    total_deleted INT := 0;
BEGIN
    LOOP
        DELETE FROM logs
        WHERE id IN (
            SELECT id FROM logs
            WHERE created_at < '2025-01-01'
            LIMIT 10000
        );

        GET DIAGNOSTICS deleted_count = ROW_COUNT;
        total_deleted := total_deleted + deleted_count;

        EXIT WHEN deleted_count = 0;

        -- Pause để tránh saturate I/O, cho autovacuum chạy
        PERFORM pg_sleep(0.1);  -- 100ms pause giữa các chunks

        RAISE NOTICE 'Deleted: % (total: %)', deleted_count, total_deleted;
    END LOOP;

    RAISE NOTICE 'Done! Total deleted: %', total_deleted;
END $$;
```

Vacuum & Bloat Management I A

Hiểu MVCC và bloat

```
PostgreSQL MVCC (Multi-Version Concurrency Control):
  UPDATE không modify row tại chỗ.
  Thay vào đó:
  1. Đánh dấu old row là "dead" (xmax timestamp)
  2. Insert new row version

  SELECT * FROM users WHERE id = 1:
  Thấy row phù hợp với transaction snapshot hiện tại
  Dead rows vẫn còn đó, chiếm disk space = BLOAT

VACUUM:
  - Đánh dấu dead tuples là reusable space
  - Không shrink file (chỉ reuse space cho inserts mới)
  - Không lock table (concurrent với reads/writes)

VACUUM FULL:
  - Rewrite table hoàn toàn → shrink file
  - LOCK TABLE exclusive suốt quá trình
  - Dùng cho emergency chỉ

pg_repack (extension):
  - Rewrite table như VACUUM FULL nhưng KHÔNG lock
  - Safe cho production
```

```sql
-- Xem bloat trên tables
SELECT
    schemaname,
    tablename,
    pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) AS total_size,
    pg_size_pretty(pg_relation_size(schemaname||'.'||tablename)) AS table_size,
    n_dead_tup,
    n_live_tup,
    ROUND(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 2) AS dead_ratio_pct,
    last_vacuum,
    last_autovacuum,
    last_analyze
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 20;

-- Dead ratio > 20%: table cần VACUUM
-- Dead ratio > 50%: autovacuum bị overwhelmed, cần investigation
```

Autovacuum tuning cho high-write tables: A

```sql
-- Autovacuum defaults quá conservative cho large tables
-- Default: vacuum khi n_dead_tup > 50 + 0.2 × n_live_tup
-- Table 10M rows: vacuum threshold = 50 + 0.2 × 10M = 2,000,050 dead tuples!
-- = cho phép 2M dead tuples trước khi vacuum

-- Aggressive autovacuum per-table (không cần restart PG)
ALTER TABLE high_write_table SET (
    autovacuum_vacuum_scale_factor = 0.01,    -- vacuum khi 1% dead (thay vì 20%)
    autovacuum_analyze_scale_factor = 0.005,  -- analyze khi 0.5% change
    autovacuum_vacuum_cost_delay = 2,         -- ms delay giữa các vacuum pages (default 20ms)
    -- Giảm cost_delay = vacuum nhanh hơn nhưng tốn I/O hơn
    autovacuum_vacuum_threshold = 100         -- minimum 100 dead tuples (thay vì 50)
);

-- postgresql.conf: global tuning
# autovacuum_max_workers = 6          # default 3, tăng cho write-heavy
# autovacuum_vacuum_cost_delay = 2ms  # default 20ms → aggressive vacuum
# autovacuum_vacuum_scale_factor = 0.05  # 5% thay vì 20%
```

```
# pg_repack: repack table mà không lock (cài extension trước)
pg_repack -h localhost -d mydb -t bloated_table --no-order
# Tạo new table ở background, copy data, swap atomically
# Table vẫn available cho reads/writes trong suốt quá trình
```

Connection-level Tuning I A

work_mem — memory cho sort và hash operations

```
-- work_mem: RAM per sort/hash OPERATION (không phải per query, không phải per connection)
-- Default: 4MB (quá thấp cho complex queries)

-- 1 query có 3 sort operations, work_mem = 64MB:
-- Total: 3 × 64MB = 192MB cho 1 query

-- Math để tính safe work_mem:
-- max_work_mem = total_RAM × 0.25 / (max_connections × max_parallel_sorts_per_query)
-- Ví dụ: 32GB RAM, 100 connections, 4 sorts/query:
-- max_work_mem = 8GB / (100 × 4) = 20MB per sort

-- postgresql.conf:
# work_mem = 16MB  -- conservative global default

-- Per-session override cho analytical queries
SET work_mem = '256MB';
[long analytical query here]
RESET work_mem;

-- Per-function:
BEGIN;
SET LOCAL work_mem = '512MB';  -- chỉ affect transaction này
SELECT * FROM big_aggregation;
COMMIT;
-- work_mem tự reset sau COMMIT

-- Xem queries đang spill to disk (cần work_mem cao hơn):
SELECT query, calls, total_exec_time, rows,
       temp_blks_read, temp_blks_written
FROM pg_stat_statements
WHERE temp_blks_written > 0
ORDER BY temp_blks_written DESC;
-- temp_blks_written > 0 = query đang sort trên disk = tăng work_mem
```

Key postgresql.conf parameters I

```
# Memory
shared_buffers = 8GB              # 25% of total RAM
# PG's own buffer pool. Data read từ disk → cache ở đây trước.
# > 25% thường không help vì OS cũng cache (double buffering)

effective_cache_size = 24GB       # 75% of total RAM
# GỢI Ý cho query planner về total cache available (PG + OS)
# Không allocate RAM, chỉ ảnh hưởng plan decisions
# Giá trị cao hơn → planner dùng indexes nhiều hơn

work_mem = 16MB                   # per sort/hash operation
# Xem phần trên. 16MB conservative, tăng per-session khi cần

maintenance_work_mem = 1GB        # cho VACUUM, CREATE INDEX, etc.
# Cao hơn → VACUUM nhanh hơn, CREATE INDEX nhanh hơn

# WAL (Write-Ahead Log)
wal_buffers = 64MB                # default -1 (auto = 3% shared_buffers, max 16MB)
# Tăng khi write-heavy workload
```

```
checkpoint_completion_target = 0.9  # spread checkpoint I/O over 90% của checkpoint interval
max_wal_size = 4GB                  # trigger checkpoint khi WAL đạt size này
min_wal_size = 1GB

# Parallel query
max_parallel_workers_per_gather = 4  # max parallel workers per query
max_parallel_workers = 8             # total parallel workers (≤ max_worker_processes)
parallel_tuple_cost = 0.1           # lower = more likely to use parallel

# Statistics
default_statistics_target = 200   # default 100. Cao hơn = better estimates, slower ANALYZE
# Cho specific high-cardinality columns:
ALTER TABLE orders ALTER COLUMN user_id SET STATISTICS 500;
```

Time-Series Optimization I A

```
-- TimescaleDB: PostgreSQL extension cho time-series

-- Cài đặt (Ubuntu)
-- sudo apt install timescaledb-2-postgresql-16
-- timescaledb-tune --quiet --yes  # auto-tune postgresql.conf

-- Convert existing table sang hypertable
SELECT create_hypertable('metrics', 'time',
    chunk_time_interval ⇒ INTERVAL '1 day'  -- partition theo ngày
);

-- TimescaleDB tự động tạo partitions (chunks) theo time interval
-- Mỗi chunk = separate internal table với own indexes
-- Query pruning: WHERE time > NOW() - INTERVAL '7 days' → chỉ scan 7 chunks

-- Native compression (95% reduction)
ALTER TABLE metrics SET (
    timescaledb.compress,
    timescaledb.compress_orderby = 'time DESC',
    timescaledb.compress_segmentby = 'device_id'
);

-- Compress chunks older than 7 days
SELECT add_compression_policy('metrics', INTERVAL '7 days');

-- Retention policy: auto-drop chunks older than 1 year
SELECT add_retention_policy('metrics', INTERVAL '1 year');

-- Continuous aggregates (materialized views với auto-refresh)
CREATE MATERIALIZED VIEW metrics_hourly
WITH (timescaledb.continuous) AS
SELECT
    time_bucket('1 hour', time) AS bucket,
    device_id,
    AVG(value) AS avg_value,
    MAX(value) AS max_value,
    MIN(value) AS min_value,
    COUNT(*) AS sample_count
FROM metrics
GROUP BY 1, 2;

-- Auto-refresh every 30 minutes
SELECT add_continuous_aggregate_policy('metrics_hourly',
    start_offset ⇒ INTERVAL '3 hours',
    end_offset   ⇒ INTERVAL '1 hour',
    schedule_interval ⇒ INTERVAL '30 minutes'
);
```

Monitoring & Diagnostics ℹ

```
-- Top queries by total time (cần pg_stat_statements extension)
-- postgresql.conf: shared_preload_libraries = 'pg_stat_statements'
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

SELECT
    LEFT(query, 100) AS query_preview,
    calls,
    ROUND(total_exec_time::numeric, 2) AS total_ms,
    ROUND(mean_exec_time::numeric, 2) AS avg_ms,
    ROUND(stddev_exec_time::numeric, 2) AS stddev_ms,
    rows,
    ROUND(100.0 * shared_blks_hit / NULLIF(shared_blks_hit + shared_blks_read, 0), 2) AS cache_hi
t_pct
FROM pg_stat_statements
WHERE calls > 100  -- chỉ xem queries đủ sample
ORDER BY total_exec_time DESC
LIMIT 20;

-- Cache hit ratio toàn bộ DB
SELECT
    SUM(heap_blks_hit) AS heap_hits,
    SUM(heap_blks_read) AS heap_reads,
    ROUND(100.0 * SUM(heap_blks_hit) / NULLIF(SUM(heap_blks_hit) + SUM(heap_blks_read), 0), 2)
AS cache_hit_pct
FROM pg_statio_user_tables;
-- Target: > 99%. Nếu < 90% → shared_buffers quá nhỏ

-- Lock waits (performance killer)
SELECT
    blocked_locks.pid AS blocked_pid,
    blocked_activity.usename AS blocked_user,
    blocking_locks.pid AS blocking_pid,
    blocking_activity.usename AS blocking_user,
    blocked_activity.query AS blocked_statement,
    blocking_activity.query AS current_statement_in_blocking_process
FROM pg_catalog.pg_locks AS blocked_locks
JOIN pg_catalog.pg_stat_activity AS blocked_activity ON blocked_activity.pid = blocked_locks.pid
JOIN pg_catalog.pg_locks AS blocking_locks ON (
    blocking_locks.locktype = blocked_locks.locktype
    AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
    AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
    AND blocking_locks.pid != blocked_locks.pid
)
JOIN pg_catalog.pg_stat_activity AS blocking_activity ON blocking_activity.pid = blocking_locks.p
id
WHERE NOT blocked_locks.GRANTED;

-- Long running queries
SELECT pid, now() - query_start AS duration, query, state
FROM pg_stat_activity
WHERE state != 'idle' AND query_start < NOW() - INTERVAL '5 minutes'
ORDER BY duration DESC;

-- Kill stuck query
SELECT pg_cancel_backend(<pid>);    -- gentle: send SIGINT, query can rollback
SELECT pg_terminate_backend(<pid>); -- force: send SIGTERM, connection dropped
```

Quick Reference — Optimization Decision Tree

```
Query chậm?
├── EXPLAIN ANALYZE → Seq Scan trên large table?
│   ├── Selectivity < 5%? → Thêm index
│   ├── Toàn bộ table? → Có thể cần partitioning
```

```
│   └── Statistics cũ? → ANALYZE table
│
├── Index Scan nhưng vẫn chậm?
│   ├── Nhiều heap lookups? → Covering index (INCLUDE)
│   ├── Kết quả lớn nhưng chỉ cần vài columns? → Select specific columns
│   └── Sort ngoài index? → Include sort column trong index
│
├── Connection timeout?
│   └── Kiểm tra PgBouncer pool size, tăng default_pool_size
│
├── High CPU?
│   ├── Hash Join trên huge tables → tăng work_mem
│   └── Many parallel queries → tune max_parallel_workers
│
├── High disk I/O?
│   ├── Cache hit < 99% → tăng shared_buffers
│   ├── Bloat cao → trigger VACUUM hoặc pg_repack
│   └── WAL chậm → tăng wal_buffers, checkpoint settings
│
└── Tất cả reads chậm?
    ├── > 90% load là reads → Add read replica
    └── Aggregation queries → Materialized views
```

Checklist Production PostgreSQL

```
Setup:
☐ PgBouncer transaction pooling giữa app và DB
☐ max_connections phù hợp với cores (100-300 thường đủ)
☐ shared_buffers = 25% RAM
☐ effective_cache_size = 75% RAM
☐ pg_stat_statements enabled

Indexing:
☐ EXPLAIN ANALYZE mọi slow query
☐ Partial indexes cho filtered queries
☐ Covering indexes cho high-frequency lookups
☐ Audit unused indexes hàng tháng (idx_scan = 0)
☐ Đo write amplification trên write-heavy tables

Maintenance:
☐ Autovacuum đang chạy (không bao giờ disable!)
☐ Autovacuum scale_factor tuned cho large tables
☐ Bloat monitoring (dead_ratio < 20%)
☐ ANALYZE sau bulk data operations

Scaling:
☐ Read replica cho read-heavy workload
☐ Partitioning cho time-series hoặc tables > 100M rows
☐ Materialized views cho expensive dashboard queries
☐ Connection pooling monitoring (cl_waiting = 0)

Monitoring:
☐ pg_stat_statements: top queries by total_time
☐ Cache hit ratio > 99%
☐ Replication lag < 10s
☐ Lock waits alert
☐ Long-running queries alert (> 5 phút)
```

21

Application-Level Patterns — Code patterns giúp 1 process xử lý 100K req/sec

> Tối ưu ở tầng infrastructure chỉ giúp được một phần. Tối ưu ở tầng application mới là đòn bẩy thực sự. Các pattern dưới đây áp dụng được bất kể bạn dùng cloud nào, server bao nhiêu tiền.

Event Loop & Non-blocking I/O

Tại sao Node.js / Go xử lý nhiều connection hơn Java / PHP trên cùng một lượng RAM?

Câu trả lời nằm ở mô hình concurrency. Java truyền thống và PHP-FPM dùng **thread-per-request**: mỗi request tạo ra (hoặc mượn từ pool) một OS thread. OS thread tốn kém — stack mặc định 1MB, context switching tốn CPU cycles.

Node.js và Go dùng mô hình khác.

Node.js — Single Thread + Event Loop

- 1 process Node.js xử lý 50K+ concurrent connections với ~100MB RAM
- Bí quyết: I/O operations (network, disk) không block event loop — chúng được giao cho OS kernel (epoll trên Linux, kqueue trên macOS)
- Khi I/O xong, kernel notify event loop, callback được đưa vào queue và execute

```
// BAD: block event loop — 1 request làm chậm tất cả
app.get('/data', (req, res) => {
  const result = fs.readFileSync('/large-file.csv'); // blocking!
  res.send(result);
});

// GOOD: non-blocking — event loop tiếp tục xử lý requests khác
app.get('/data', async (req, res) => {
  const result = await fs.promises.readFile('/large-file.csv'); // async
```

```
  res.send(result);
});
```

```
# Tune Node.js thread pool cho I/O-heavy workloads
UV_THREADPOOL_SIZE=16 node server.js

# Monitor event loop lag (>50ms = có blocking code)
node --prof server.js
node --prof-process isolate-*.log | grep "Bottom up"
```

Go — Goroutines (M:N Threading)

```
Goroutine 1 (2KB stack)  ─┐
Goroutine 2 (2KB stack)  ─┤
...                       ├─► Go Scheduler (M:N) ──► OS Threads (= CPU cores)
Goroutine 999999 (2KB)   ─┘
```

```
// 1 goroutine = 2KB stack ban đầu (tự động grow đến GB nếu cần)
// 1 OS thread = 1MB stack cố định

// Math:
// 1000 OS threads  = 1000 × 1MB  = 1GB RAM (chỉ cho stacks)
// 1M goroutines    = 1M × 2KB    = 2GB RAM (cho toàn bộ stacks)

// Spawn 1M goroutines — hoàn toàn khả thi
for i := 0; i < 1_000_000; i++ {
    go handleConnection(conn)
}
```

```
// HTTP server Go: mỗi connection = 1 goroutine
// Runtime scheduler multiplex goroutines lên OS threads
// I/O blocking → goroutine suspend, OS thread free cho goroutine khác
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    // Gọi database — goroutine suspend khi chờ network
    // OS thread KHÔNG bị giữ — tiếp tục chạy goroutine khác
    result, err := db.QueryContext(r.Context(), "SELECT ...")
})
```

Python asyncio — Event Loop + Coroutines

```
import asyncio
import uvloop  # drop-in replacement, 2x faster than default

# uvloop dùng libuv (cùng library với Node.js)
asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())

async def handle_request(reader, writer):
    data = await reader.read(1024)      # coroutine suspend — event loop làm việc khác
    response = await process(data)      # coroutine suspend
    writer.write(response)
    await writer.drain()                # coroutine suspend

# asyncio.gather: chạy nhiều coroutines concurrently trên 1 thread
async def batch_fetch(urls):
    tasks = [fetch(url) for url in urls]
    results = await asyncio.gather(*tasks)  # tất cả chạy concurrently
    return results
```

```
# uvloop install
pip install uvloop

# Benchmark: uvloop vs default asyncio
# Default asyncio: ~30K req/sec
# uvloop:          ~60K req/sec
```

```
# Node.js:        ~50K req/sec
# (Benchmark nguồn: uvloop GitHub)
```

Cảnh báo quan trọng

```
CPU-bound work BLOCKS event loop.

Event loop là single-threaded. Nếu bạn làm CPU-heavy work
(image resize, video encode, machine learning inference, bcrypt hash)
ngay trong event loop → tất cả requests khác phải chờ.

Solutions:
├── Node.js: worker_threads, cluster, child_process
├── Python:  ProcessPoolExecutor, celery workers
└── Go:      goroutines tự nhiên handle (scheduler preemptive từ Go 1.14)
```

```
// Node.js: offload CPU work sang worker thread
const { Worker } = require('worker_threads');

app.post('/resize-image', async (req, res) => {
  const result = await new Promise((resolve, reject) => {
    const worker = new Worker('./image-worker.js', {
      workerData: req.body
    });
    worker.on('message', resolve);
    worker.on('error', reject);
  });
  res.json(result);
});
```

Batching & Buffering

DataLoader Pattern — Giải quyết N+1 Problem

N+1 problem là một trong những nguyên nhân phổ biến nhất khiến API chậm không rõ lý do.

```
Scenario: Render danh sách 100 bài post, mỗi post kèm tên tác giả

WITHOUT DataLoader:
  GET /posts → SELECT * FROM posts LIMIT 100        (1 query)
  Render post 1 → SELECT * FROM users WHERE id = 5  (1 query)
  Render post 2 → SELECT * FROM users WHERE id = 12 (1 query)
  Render post 3 → SELECT * FROM users WHERE id = 5  (1 query, duplicate!)
  ...
  Total: 1 + 100 = 101 queries → ~50ms × 101 = 5050ms ≈ 5 seconds

WITH DataLoader:
  GET /posts → SELECT * FROM posts LIMIT 100        (1 query)
  DataLoader collects all user_ids: [5, 12, 5, 8, ...]
  DataLoader deduplicates: [5, 8, 12, ...]
  SELECT * FROM users WHERE id IN (5, 8, 12, ...)   (1 query)
  Total: 2 queries → ~100ms ≈ 0.1 seconds

Improvement: 50x faster
```

```
// Implementation: DataLoader batches requests within same event loop tick
const DataLoader = require('dataloader');

const userLoader = new DataLoader(async (userIds) => {
  // userIds = [5, 8, 12, 17, ...] — tất cả IDs collected trong 1 tick
  const users = await db.query(
    'SELECT * FROM users WHERE id = ANY($1)', [userIds]
  );
```

```
  // Quan trọng: phải return theo đúng thứ tự input
  const userMap = new Map(users.rows.map(u => [u.id, u]));
  return userIds.map(id => userMap.get(id) || null);
});

// Mỗi resolver gọi load() — DataLoader tự động batch
const resolvers = {
  Post: {
    author: (post) => userLoader.load(post.user_id), // batched!
  }
};
```

```
# Python: strawberry-graphql có built-in DataLoader
from strawberry.dataloader import DataLoader

async def load_users(keys: list[int]) -> list[User]:
    users = await db.fetch_all(
        "SELECT * FROM users WHERE id = ANY($1)", keys
    )
    user_map = {u.id: u for u in users}
    return [user_map.get(key) for key in keys]

UserLoader = DataLoader(load_fn=load_users)
```

Write Buffering — Batch INSERT

```
Math:
  1000 individual INSERTs:
    1000 round trips × 0.5ms avg latency = 500ms
    + 1000 × parse/plan/execute overhead ≈ 600ms total

  1 batch INSERT (1000 rows):
    1 round trip × 0.5ms + bulk overhead ≈ 5-15ms total

  Improvement: 40-120x faster
```

```
-- BAD: 1000 individual inserts
INSERT INTO events (user_id, type, ts) VALUES (1, 'click', NOW());
INSERT INTO events (user_id, type, ts) VALUES (2, 'view', NOW());
-- ... 998 more

-- GOOD: 1 batch insert
INSERT INTO events (user_id, type, ts) VALUES
  (1, 'click', NOW()),
  (2, 'view', NOW()),
  -- ... 998 more
  (1000, 'purchase', NOW());
```

```
// Go: batch writer với ticker
type BatchWriter struct {
    buf     []Event
    mu      sync.Mutex
    flushCh chan struct{}
}

func (bw *BatchWriter) Add(e Event) {
    bw.mu.Lock()
    bw.buf = append(bw.buf, e)
    if len(bw.buf) >= 500 {
        bw.flush() // flush khi đầy
    }
    bw.mu.Unlock()
}

func (bw *BatchWriter) Start() {
    ticker := time.NewTicker(100 * time.Millisecond)
    for range ticker.C {
        bw.mu.Lock()
```

```
        bw.flush() // flush mỗi 100ms dù chưa đầy
        bw.mu.Unlock()
    }
}
```

Log & Metrics Buffering

```
# BAD: mỗi log line = 1 write syscall
import logging
logging.info("Request processed") # syscall per line → I/O overhead

# GOOD: buffer logs, flush theo batch
import logging
handler = logging.handlers.MemoryHandler(
    capacity=1000,        # flush khi đủ 1000 entries
    flushLevel=logging.ERROR,
    target=file_handler,
    flushOnClose=True
)
# Hoặc dùng async logging handler
```

```
# Metrics: aggregate client-side, push theo interval
class MetricsBuffer:
    def __init__(self, flush_interval=10):
        self.counters = defaultdict(int)
        self.gauges = {}
        self._start_flush_loop(flush_interval)

    def increment(self, metric, value=1):
        self.counters[metric] += value  # local aggregation, no network

    def _flush(self):
        # Push tất cả metrics 1 lần mỗi 10 giây
        payload = {**self.counters, **self.gauges}
        requests.post('http://metrics-server/push', json=payload)
        self.counters.clear()
```

Cảnh báo

```
Buffering = đánh đổi giữa performance và durability.

Nếu process crash trước khi flush:
  - Write buffer: mất tối đa flush_interval × write_rate records
  - Log buffer: mất logs trong buffer (debug blind spot)
  - Metrics buffer: gap trong time series

Tune flush interval dựa trên acceptable data loss window:
  - Transactional data (orders, payments): flush every write hoặc WAL-based
  - Analytics events: flush every 1-5 seconds, acceptable to lose < 5s
  - Metrics: flush every 10-60 seconds, acceptable
  - Logs: flush every 100ms-1s với MemoryHandler
```

Compression

So sánh Algorithms

```
Algorithm   Ratio      Speed (compress)   CPU cost   Best for
──────────────────────────────────────────────────────────────
gzip        ~70%       ~250 MB/s          Low        Text, HTML, JSON, CSS, JS
Brotli      ~75-78%    ~100 MB/s          Medium     Pre-compressed static assets
zstd        ~72%       ~500 MB/s          Low        Dynamic content, logs
```

```
lz4         ~50%       ~700 MB/s          Very low   Real-time streaming
snappy      ~45%       ~1 GB/s            Very low   Internal microservice calls
```

Nginx Compression Config

```
# /etc/nginx/nginx.conf hoặc server block
http {
    # Bật gzip
    gzip on;
    gzip_vary on;
    gzip_proxied any;
    gzip_comp_level 4;         # Sweet spot: level 1-4
                               # Level 1: fastest, ~60% reduction
                               # Level 4: ~70% reduction, 2x CPU vs level 1
                               # Level 9: ~73% reduction, 10x CPU vs level 1
                               # Beyond 4: diminishing returns
    gzip_min_length 1000;      # Đừng compress files nhỏ hơn 1KB
    gzip_types
        text/plain
        text/css
        text/javascript
        application/javascript
        application/json
        application/xml
        image/svg+xml;         # SVG là text, compressible

    # Pre-compressed static assets (Brotli)
    # Build step: brotli --quality=11 style.css → style.css.br
    brotli on;
    brotli_static on;          # Serve .br file nếu tồn tại
    brotli_comp_level 6;
    brotli_types text/css application/javascript application/json;
}
```

Bandwidth Math

```
Scenario: API serving JSON responses

Without compression:
  1M users/day × 100KB average response = 100GB bandwidth/day
  At $0.09/GB (AWS) = $9/day = $270/month

With gzip (70% reduction, response → 30KB):
  1M users/day × 30KB = 30GB bandwidth/day
  Cost = $2.70/day = $81/month
  Saving: $189/month

With Brotli (75% reduction, response → 25KB):
  1M users/day × 25KB = 25GB bandwidth/day
  Cost = $2.25/day = $67.50/month
  Saving: $202.50/month
```

Binary Serialization — Protocol Buffers

```
JSON   1000 bytes → {"id":123,"name":"Nguyen Van A","email":"test@example.com",...}
Protobuf 300 bytes → binary encoding (field tags + varint + length-prefixed strings)

Improvement: 70% smaller payload + 5-10x faster parsing

Math:
  10K API calls/sec × 1KB JSON = 10MB/sec serialization/deserialization
  10K API calls/sec × 300B Protobuf = 3MB/sec
  + Protobuf parsing 5x faster → CPU freed for actual business logic
```

```
// user.proto
syntax = "proto3";
```

```
message User {
    int32 id = 1;
    string name = 2;
    string email = 3;
    int64 created_at = 4;
}
```

```
# Generate code
protoc --go_out=. --go-grpc_out=. user.proto      # Go
protoc --python_out=. user.proto                   # Python
protoc --js_out=. user.proto                       # JavaScript
```

Cảnh báo

```
Đừng compress những thứ đã compressed:
  - JPEG, PNG, WebP, AVIF → ảnh đã nén, compress thêm = tốn CPU, file lớn hơn
  - MP4, WebM, AAC, MP3   → video/audio đã nén
  - ZIP, GZ, BR files     → đã nén sẵn

Compression tăng latency cho streaming responses:
  - SSE (Server-Sent Events): compressor cần buffer → tăng latency mỗi event
  - WebSocket real-time updates: disable per-message compression nếu latency critical
  - Use permessage-deflate cho WebSocket chỉ khi bandwidth quan trọng hơn latency
```

Pagination & Lazy Loading

Vấn đề với Offset Pagination

```
-- OFFSET pagination — phổ biến nhưng tệ ở deep pages
SELECT * FROM posts ORDER BY created_at DESC LIMIT 20 OFFSET 10000;

-- Database phải:
-- 1. Scan và đọc 10020 rows
-- 2. Throw đi 10000 rows đầu
-- 3. Return 20 rows cuối
-- Cost = O(OFFSET) — linear với page depth
-- Page 1:    OFFSET 0     → fast (< 1ms)
-- Page 500:  OFFSET 10000 → slow (50-200ms)
-- Page 5000: OFFSET 100000 → very slow (seconds)
```

Cursor-Based Pagination

```
-- First page
SELECT * FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 20;
-- Returns: last row có created_at='2024-01-15 10:30:00', id=12345

-- Next page — dùng cursor thay vì OFFSET
SELECT * FROM posts
WHERE (created_at, id) < ('2024-01-15 10:30:00', 12345)
ORDER BY created_at DESC, id DESC
LIMIT 20;
-- Cost = O(1) với index — nhanh như page 1 dù bạn ở page 5000
```

```
// API Response với cursor
{
  "data": [...],
  "pagination": {
    "next_cursor": "eyJjcmVhdGVkX2F0IjoiMjAyNC0wMS0xNSIsImlkIjoxMjM0NX0=",
```

```
    "has_more": true
  }
}

// Client request trang tiếp
GET /api/posts?cursor=eyJjcmVhdGVkX2F0IjoiMjAyNC0wMS0xNSIsImlkIjoxMjM0NX0=&limit=20
```

```python
import base64, json

def encode_cursor(created_at, post_id):
    payload = {"created_at": created_at.isoformat(), "id": post_id}
    return base64.b64encode(json.dumps(payload).encode()).decode()

def decode_cursor(cursor_str):
    payload = json.loads(base64.b64decode(cursor_str))
    return payload["created_at"], payload["id"]

async def get_posts(cursor=None, limit=20):
    if cursor:
        created_at, last_id = decode_cursor(cursor)
        rows = await db.fetch_all(
            """SELECT * FROM posts
               WHERE (created_at, id) < ($1, $2)
               ORDER BY created_at DESC, id DESC
               LIMIT $3""",
            created_at, last_id, limit + 1  # fetch 1 extra to check has_more
        )
    else:
        rows = await db.fetch_all(
            "SELECT * FROM posts ORDER BY created_at DESC, id DESC LIMIT $1",
            limit + 1
        )

    has_more = len(rows) > limit
    data = rows[:limit]
    next_cursor = encode_cursor(data[-1].created_at, data[-1].id) if has_more else None
    return {"data": data, "next_cursor": next_cursor, "has_more": has_more}
```

GraphQL Sparse Fieldsets

```
# Client chỉ request những fields cần thiết
# Không over-fetching như REST API trả về toàn bộ object

# BAD: REST trả về user object 50 fields
GET /api/users/123
# → { id, name, email, phone, address, bio, avatar, settings, preferences, ... }
# Client chỉ cần id + name → 80% data bị lãng phí

# GOOD: GraphQL chỉ fetch cần thiết
query {
  user(id: 123) {
    id
    name
  }
}
# → { id: 123, name: "Nguyen Van A" } — chính xác, không hơn
```

```python
# REST API: sparse fieldsets via query param
GET /api/users/123?fields=id,name,email

# Implementation (FastAPI)
@app.get("/users/{user_id}")
async def get_user(user_id: int, fields: str = None):
    user = await db.get_user(user_id)
    if fields:
        allowed = set(fields.split(','))
        return {k: v for k, v in user.dict().items() if k in allowed}
    return user
```

Cảnh báo

```
Offset pagination + millions of rows = death by scanning.

Real case: E-commerce với 10M products.
  - Page 500 (OFFSET 10000): 200ms → user frustrated
  - Page 5000 (OFFSET 100000): 3 seconds → user leaves
  - Page 50000 (OFFSET 1000000): database timeout → error

Always dùng cursor pagination cho:
  - Infinite scroll
  - "Load more" buttons
  - Any dataset > 10K rows
  - Any pagination past page 10
```

Rate Limiting Patterns

Token Bucket — Cho phép Bursts

```
Token Bucket concept:
  - Bucket capacity: 100 tokens (max burst size)
  - Refill rate: 10 tokens/second
  - Each request costs 1 token

  t=0:   bucket=100, user sends 80 requests → bucket=20 (allowed)
  t=1:   bucket=30 (refilled 10), user sends 20 → bucket=10 (allowed)
  t=2:   bucket=20 (refilled 10), user sends 25 → 20 allowed, 5 rejected (429)
```

```lua
-- Redis Lua script (atomic — không race condition)
-- KEYS[1] = rate_limit:{user_id}
-- ARGV[1] = capacity, ARGV[2] = refill_rate, ARGV[3] = current_time

local key = KEYS[1]
local capacity = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4])

local bucket = redis.call('HMGET', key, 'tokens', 'last_refill')
local tokens = tonumber(bucket[1]) or capacity
local last_refill = tonumber(bucket[2]) or now

-- Refill tokens based on elapsed time
local elapsed = now - last_refill
local refill = math.floor(elapsed * refill_rate)
tokens = math.min(capacity, tokens + refill)

if tokens ⩾ requested then
    tokens = tokens - requested
    redis.call('HMSET', key, 'tokens', tokens, 'last_refill', now)
    redis.call('EXPIRE', key, math.ceil(capacity / refill_rate) + 1)
    return 1  -- allowed
else
    redis.call('HMSET', key, 'tokens', tokens, 'last_refill', now)
    return 0  -- rejected
end
```

```python
# Python integration
import redis
import time

r = redis.Redis()
RATE_LIMIT_SCRIPT = r.register_script(LUA_SCRIPT)
```

```
def is_allowed(user_id: str, capacity=100, refill_rate=10, cost=1) -> bool:
    result = RATE_LIMIT_SCRIPT(
        keys=[f"rate_limit:{user_id}"],
        args=[capacity, refill_rate, time.time(), cost]
    )
    return bool(result)

@app.middleware("http")
async def rate_limit_middleware(request: Request, call_next):
    user_id = get_user_id(request)
    if not is_allowed(user_id):
        return Response(
            status_code=429,
            headers={"Retry-After": "1"},
            content="Rate limit exceeded"
        )
    return await call_next(request)
```

Sliding Window — Chính xác hơn Fixed Window

```
Fixed Window edge case:
  Window: 100 req/minute
  0:59 — user sends 100 requests → allowed (window 0:00-1:00)
  1:01 — user sends 100 requests → allowed (new window 1:00-2:00)
  Thực tế: 200 requests trong 2 giây — gấp đôi giới hạn!

Sliding Window không có edge case này:
  At any point in time, chỉ đếm requests trong 60 giây gần nhất
```

```
# Redis Sorted Set: sliding window
import redis
import time

r = redis.Redis()

def sliding_window_check(user_id: str, limit=100, window_seconds=60) -> bool:
    key = f"sliding:{user_id}"
    now = time.time()
    window_start = now - window_seconds

    pipe = r.pipeline()
    # Xóa entries cũ hơn window
    pipe.zremrangebyscore(key, '-inf', window_start)
    # Đếm requests trong window
    pipe.zcard(key)
    # Thêm request hiện tại
    pipe.zadd(key, {str(now): now})
    # Set TTL
    pipe.expire(key, window_seconds + 1)

    results = pipe.execute()
    request_count = results[1]

    return request_count < limit
```

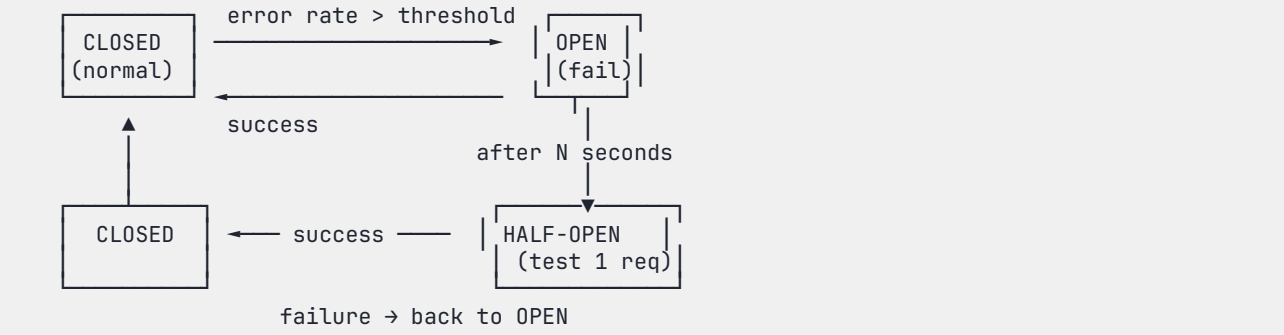
Cảnh báo

```
Fixed window (e.g., 100 req/min reset tại :00) có boundary burst:
  - 100 req tại 0:59 → allowed
  - 100 req tại 1:00 → allowed (new window)
  - Result: 200 requests trong 1 giây → có thể crash downstream

Adaptive Rate Limiting — tự động điều chỉnh theo load:
  - Healthy: giữ limit bình thường
  - High load (CPU > 80%): giảm limit 50%
  - Overloaded: return 503 với Retry-After
  Pattern này kết hợp tốt với circuit breaker
```

```go
// Client-side: exponential backoff với jitter khi nhận 429
func backoffRetry(fn func() error) error {
    baseDelay := 100 * time.Millisecond
    for attempt := 0; attempt < 5; attempt++ {
        err := fn()
        if err == nil {
            return nil
        }
        if !isRateLimited(err) {
            return err
        }
        // Exponential backoff: 100ms, 200ms, 400ms, 800ms, 1600ms
        delay := baseDelay * time.Duration(1<<attempt)
        // Jitter: ±25% để tránh thundering herd
        jitter := time.Duration(rand.Int63n(int64(delay / 4)))
        time.Sleep(delay + jitter)
    }
    return fmt.Errorf("max retries exceeded")
}
```

Graceful Degradation A

Circuit Breaker Pattern




```python
import time
from enum import Enum
from threading import Lock

class State(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    def __init__(self, failure_threshold=5, recovery_timeout=30, success_threshold=2):
        self.failure_threshold = failure_threshold   # lỗi liên tiếp để open
        self.recovery_timeout = recovery_timeout     # giây trước khi thử lại
        self.success_threshold = success_threshold   # success liên tiếp để close
        self.state = State.CLOSED
        self.failure_count = 0
        self.success_count = 0
        self.last_failure_time = None
        self._lock = Lock()

    def call(self, fn, fallback=None):
        with self._lock:
            if self.state == State.OPEN:
                if time.time() - self.last_failure_time > self.recovery_timeout:
                    self.state = State.HALF_OPEN
                    self.success_count = 0
```

```
                else:
                    # Trả về fallback thay vì gọi service
                    return fallback() if fallback else None

        try:
            result = fn()
            with self._lock:
                if self.state == State.HALF_OPEN:
                    self.success_count += 1
                    if self.success_count >= self.success_threshold:
                        self.state = State.CLOSED
                        self.failure_count = 0
            return result
        except Exception as e:
            with self._lock:
                self.failure_count += 1
                self.last_failure_time = time.time()
                if self.failure_count >= self.failure_threshold:
                    self.state = State.OPEN
            raise

# Usage
db_breaker = CircuitBreaker(failure_threshold=5, recovery_timeout=30)

async def get_user(user_id):
    def db_call():
        return db.query("SELECT * FROM users WHERE id = $1", user_id)

    def fallback():
        return cache.get(f"user:{user_id}")  # serve stale data

    return db_breaker.call(db_call, fallback=fallback)
```

Timeout Cascading

```
API Gateway timeout:  30 seconds  ← client-facing
    └── Service A timeout: 10 seconds
            └── Service B timeout: 5 seconds
                    └── Database timeout: 3 seconds

Rule: mỗi layer phải timeout TRƯỚC caller của nó.
Nếu Service A timeout = 30s (bằng Gateway):
  - Gateway timeout trước khi nhận response từ A
  - A vẫn đang chạy, tốn resources vô ích
  - Resource leak + cascading failures
```

```
// Go: context-based timeout cascading
func (h *Handler) GetUserOrders(w http.ResponseWriter, r *http.Request) {
    // Context từ HTTP request có timeout 30s từ gateway
    ctx := r.Context()

    // Service-level timeout: 10s (ngắn hơn gateway)
    ctx, cancel := context.WithTimeout(ctx, 10*time.Second)
    defer cancel()

    user, err := h.userService.GetUser(ctx, userID)
    if err != nil {
        // Context timeout → trả về cached data
        if errors.Is(err, context.DeadlineExceeded) {
            user = h.cache.GetUser(userID) // fallback
        }
    }
}

// Database layer: 3s timeout
func (s *UserService) GetUser(ctx context.Context, id int) (*User, error) {
    dbCtx, cancel := context.WithTimeout(ctx, 3*time.Second)
    defer cancel()
```

```
    return s.db.QueryRowContext(dbCtx, "SELECT * FROM users WHERE id = $1", id)
}
```

Load Shedding

```python
import psutil

class LoadShedder:
    def __init__(self, cpu_threshold=80, memory_threshold=90):
        self.cpu_threshold = cpu_threshold
        self.memory_threshold = memory_threshold

    def should_shed(self, request_priority: str) -> bool:
        cpu = psutil.cpu_percent(interval=0.1)
        memory = psutil.virtual_memory().percent

        if cpu > self.cpu_threshold or memory > self.memory_threshold:
            # Shed low-priority requests khi overloaded
            if request_priority == "low":
                return True
        return False

shedder = LoadShedder()

@app.middleware("http")
async def load_shedding_middleware(request: Request, call_next):
    priority = request.headers.get("X-Request-Priority", "low")
    if shedder.should_shed(priority):
        return Response(
            status_code=503,
            headers={"Retry-After": "5"},
            content="Server overloaded, please retry"
        )
    return await call_next(request)
```

Worker Pattern ℹ

Background Job Processing

```
Synchronous (BAD for slow work):
  Client → API → [process email + resize image + send notification] → Response
  Wait time: 2-30 seconds → bad UX, ties up API thread

Async Worker Pattern (GOOD):
  Client → API → enqueue job → Response (202 Accepted) ← immediate
                    │
                    ▼
                ┌─────────┐
                │  Queue  │
                └─────────┘
                    │
                    ▼
                ┌─────────┐
                │ Workers │ ← process at own pace
                └─────────┘
```

```python
# FastAPI + Redis Queue (RQ)
from rq import Queue
from redis import Redis

redis_conn = Redis()
q = Queue(connection=redis_conn)

@app.post("/users/{user_id}/send-report")
async def send_report(user_id: int):
    # Enqueue — trả về ngay, không chờ job xong
```

```
    job = q.enqueue(
        generate_and_send_report,  # function này chạy trong worker
        user_id,
        job_timeout=300,           # 5 minutes max
        result_ttl=3600            # keep result 1 hour
    )
    return {"job_id": job.id, "status": "queued"}  # 202 response

@app.get("/jobs/{job_id}")
async def get_job_status(job_id: str):
    job = Job.fetch(job_id, connection=redis_conn)
    return {
        "status": job.get_status(),  # queued/started/finished/failed
        "result": job.result,
        "error": job.exc_info
    }
```

```
# Start workers
rq worker --with-scheduler default  # single worker
rq worker --burst default           # process all then exit (CI/batch mode)

# Worker concurrency: match to task type
# CPU-bound (image resize): workers = CPU cores (4 core → 4 workers)
# I/O-bound (email send):   workers = 10-50 × CPU cores
```

Idempotency — Jobs phải an toàn khi retry

```
# BAD: không idempotent — nếu retry, gửi email 2 lần
def send_welcome_email(user_id):
    user = db.get_user(user_id)
    email_service.send(user.email, "Welcome!")

# GOOD: idempotent — check trước khi gửi
def send_welcome_email(user_id):
    # Dùng idempotency key trong DB
    lock_key = f"welcome_email:{user_id}"

    # Redis SET NX (Only set if Not eXists) — atomic
    acquired = redis.set(lock_key, "1", nx=True, ex=86400)  # expire 24h
    if not acquired:
        return  # Đã gửi rồi, skip

    user = db.get_user(user_id)
    email_service.send(user.email, "Welcome!")
```

Dead Letter Queue

```
# Sau N lần fail → DLQ để manual inspect
from rq import Queue
from rq.job import Job

failed_queue = Queue('failed', connection=redis_conn)

# Tự động: RQ move failed jobs sang failed queue sau max_retries
q = Queue(connection=redis_conn)
q.enqueue(
    process_payment,
    payment_id,
    retry=Retry(max=3, interval=[10, 30, 60])  # retry sau 10s, 30s, 60s
)

# Manual: inspect và retry từ DLQ
for job in failed_queue.jobs:
    print(f"Job {job.id} failed: {job.exc_info}")
    # Sau khi fix bug, requeue
    job.requeue()
```

Server-Sent Events vs WebSocket vs Long Polling B

So sánh

Method	Direction	Complexity	Overhead	Use Case
Long Polling	Server→Client	Low	High	Đơn giản, < 1K users
SSE	Server→Client	Low	Low	Notifications, feeds
WebSocket	Bidirectional	Medium	Low	Chat, real-time collab
WebRTC	P2P	High	Very Low	Video/audio calls

SSE — Đơn giản và Hiệu quả

```python
# FastAPI SSE endpoint
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import asyncio, json

@app.get("/notifications")
async def notifications(user_id: int):
    async def event_generator():
        while True:
            # Check for new notifications
            events = await get_pending_notifications(user_id)
            for event in events:
                # SSE format: "data: ...\n\n"
                yield f"data: {json.dumps(event)}\n\n"
            await asyncio.sleep(1)  # poll mỗi 1 giây

    return StreamingResponse(
        event_generator(),
        media_type="text/event-stream",
        headers={
            "Cache-Control": "no-cache",
            "X-Accel-Buffering": "no",  # tắt Nginx buffering
        }
    )
```

```javascript
// Client: SSE tự động reconnect
const evtSource = new EventSource('/notifications');

evtSource.onmessage = (event) => {
    const data = JSON.parse(event.data);
    showNotification(data);
};

evtSource.onerror = () => {
    // Browser tự động reconnect sau 3 giây
    console.log('Connection lost, browser will reconnect...');
};
```

```nginx
# Nginx: tắt buffering cho SSE
location /notifications {
    proxy_pass http://app:8000;
    proxy_buffering off;          # quan trọng!
    proxy_cache off;
    proxy_read_timeout 86400;     # 24h keepalive
    proxy_set_header Connection '';
    proxy_http_version 1.1;
    chunked_transfer_encoding on;
}
```

Cảnh báo

```
SSE + HTTP/1.1: browser limit = 6 connections per domain.
  - Tab 1 mở SSE connection 1
  - Tab 2 mở SSE connection 2
  - ...Tab 6 mở SSE connection 6
  - Tab 7: blocked — phải chờ 1 trong 6 connections đóng

Fix: dùng HTTP/2
  - HTTP/2 multiplexing: nhiều SSE streams trên 1 TCP connection
  - Browser limit = 100 concurrent streams (thực tế không bao giờ đạt)
  - Nginx: http2 on; (trong listen directive)

1M SSE connections = 1M open HTTP connections trên server:
  Cần kernel tuning:
    net.core.somaxconn = 65535
    net.ipv4.tcp_max_syn_backlog = 65535
    fs.file-max = 1000000
    ulimit -n 1000000
  Và load balancer phải support sticky sessions hoặc pub/sub backend
```

Tổng kết Performance Stack

Layer	Technique	Improvement
Concurrency Model	Go goroutines / Node async	10-100x connections/RAM
Query Optimization	DataLoader batching	50x query reduction
Write Performance	Batch INSERT	40-120x write throughput
Bandwidth	gzip/Brotli compression	60-75% bandwidth saving
Serialization	Protobuf vs JSON	70% smaller + 5x faster parse
Pagination	Cursor vs Offset	O(1) vs O(n) at deep pages
Rate Limiting	Token bucket (Redis atomic)	Consistent, burst-friendly
Resilience	Circuit breaker + timeout	Prevent cascading failures
Async Processing	Worker queue pattern	Immediate response + scale
Real-time	SSE over long polling	Lower overhead, simpler code

> Không có silver bullet. Profile trước, optimize sau. Đo lường là bắt buộc — đừng đoán.

22

Async Processing & Message Queues — Absorb traffic spikes without scaling

> Scaling server lên 2x tốn tiền gấp đôi. Thêm một queue tốn $0. Async processing là kỹ thuật có ROI cao nhất trong kiến trúc backend.

Why Async B

Synchronous Request = Chain of Failure

```
Synchronous flow:
  1M users → API → DB → API → Users
                    ↑
               DB handles 1K writes/sec
               Khi traffic = 10K writes/sec:
               → DB connection pool exhausted
               → API returns 500 errors
               → Users see errors
               → Revenue lost

Async flow:
  1M users → API → Queue → API returns 202 ← immediate!
                     ↓
                  Workers → DB (at DB's pace: 1K writes/sec)
                  → No errors, no timeouts, smooth processing
```

Spike Absorption Math

```
Giả thiết:
  DB max throughput:  1,000 writes/sec
  Normal traffic:     800 writes/sec (safe)
  Spike duration:     5 minutes (300 seconds)
  Spike intensity:    10,000 writes/sec

SYNCHRONOUS approach:
  DB overloaded at second 1
  Connection timeouts at second 5
  Users see 500 errors for entire 5-minute spike
  Messages lost unless client retries

ASYNC approach:
  Queue absorbs excess: (10,000 - 1,000) × 300 = 2,700,000 messages buffered
  Workers drain at: 1,000/sec
  Time to clear backlog: 2,700,000 / 1,000 = 2,700 seconds = 45 minutes
  User experience: immediate 202 response, zero errors
  Tradeoff: eventual processing (45 min delay) vs real-time

Kết luận: 5 phút spike → không 1 error nào. DB không bị touch quá capacity.
```

Khi nào KHÔNG dùng Async

```
Dùng synchronous khi:
  ✓ User cần kết quả ngay (search query, login, payment confirmation)
  ✓ Job cần kết quả để trả về response (generate report inline)
  ✓ Transaction cần atomic (order + inventory decrement phải cùng lúc)

Dùng async khi:
  ✓ Email, SMS, push notifications
  ✓ Image/video processing
  ✓ Report generation
  ✓ Webhook delivery
  ✓ Data sync giữa systems
  ✓ Batch operations (bulk import, bulk update)
  ✓ Any work > 200ms
```

Redis as Queue B

Simple Queue — LPUSH + BRPOP

```
# Producer: push vào queue
LPUSH jobs:email '{"user_id":123,"template":"welcome"}'

# Consumer: blocking pop (không polling, không CPU waste)
BRPOP jobs:email 0     # 0 = block forever until message available
# Returns: ["jobs:email", "{\"user_id\":123,...}"]
```

```python
import redis
import json
import time

r = redis.Redis(host='localhost', port=6379, decode_responses=True)

# Producer
def enqueue_email(user_id: int, template: str):
    job = {"user_id": user_id, "template": template, "started_at": time.time()}
    r.lpush("jobs:email", json.dumps(job))

# Consumer (run in worker process)
def process_emails():
    while True:
        # BRPOP blocks — 0 CPU usage while waiting
        _, message = r.brpop("jobs:email", timeout=0)
        job = json.loads(message)
        send_email(job["user_id"], job["template"])
        # BUG: nếu worker crash ở đây → message mất!
```

Reliable Queue — BRPOPLPUSH (Atomic)

```python
# BRPOPLPUSH: atomically pop từ source + push vào processing list
# Nếu worker crash → message vẫn còn trong processing list
def reliable_worker():
    while True:
        # Atomic: pop from jobs:email, push to jobs:email:processing
        message = r.brpoplpush("jobs:email", "jobs:email:processing", timeout=0)

        try:
            job = json.loads(message)
            send_email(job["user_id"], job["template"])
            # Success: remove from processing list
            r.lrem("jobs:email:processing", 1, message)
        except Exception as e:
            # Failure: message remains in processing list
```

```
            # Separate recovery process sẽ re-queue sau timeout
            log_error(e)

# Recovery process: chạy định kỳ
def recover_stuck_jobs():
    # Lấy tất cả jobs đang trong processing
    processing = r.lrange("jobs:email:processing", 0, -1)
    for message in processing:
        job = json.loads(message)
        # Nếu job quá cũ (stuck worker), re-queue
        if job.get("started_at", 0) < time.time() - 300:  # 5 min timeout (0 = thiếu mốc → re-
queue)
            r.lpush("jobs:email", message)
            r.lrem("jobs:email:processing", 1, message)
```

Redis Streams — Production-Grade Queue ℹ

```
# Producer: XADD
XADD email-jobs * user_id 123 template welcome created_at 1706000000

# Consumer group: mỗi message chỉ deliver tới 1 consumer trong group
XGROUP CREATE email-jobs workers $ MKSTREAM

# Consumer: XREADGROUP (acknowledge-based)
XREADGROUP GROUP workers consumer-1 COUNT 10 BLOCK 0 STREAMS email-jobs >

# Acknowledge sau khi process xong
XACK email-jobs workers 1706000000-0

# Lấy lại messages chưa ack (stuck/crashed workers)
XPENDING email-jobs workers - + 10
XCLAIM email-jobs workers consumer-1 300000 1706000000-0  # reclaim sau 5 min
```

```
# Redis Streams consumer với Python
import redis

r = redis.Redis(decode_responses=True)

def create_consumer_group():
    try:
        r.xgroup_create("email-jobs", "workers", "$", mkstream=True)
    except redis.ResponseError:
        pass  # Group already exists

def stream_worker(consumer_name: str):
    create_consumer_group()

    while True:
        # Read up to 10 messages, block 2 seconds if none
        messages = r.xreadgroup(
            "workers",
            consumer_name,
            {"email-jobs": ">"},  # ">" = only new messages
            count=10,
            block=2000
        )

        if not messages:
            # Check for stuck messages (PEL — pending entry list)
            recover_pending_messages(consumer_name)
            continue

        for stream, entries in messages:
            for entry_id, data in entries:
                try:
                    process_email(data)
                    # Acknowledge success
                    r.xack("email-jobs", "workers", entry_id)
                except Exception as e:
```

```
                    # Don't ack → message stays in PEL
                    # After N retries → move to DLQ
                    handle_failure(entry_id, data, e)

def recover_pending_messages(consumer_name: str):
    # Find messages pending > 5 minutes (likely from crashed consumer)
    pending = r.xpending_range(
        "email-jobs", "workers",
        min="-", max="+",
        count=10,
        consumername=None  # all consumers
    )

    for entry in pending:
        if entry["time_since_delivered"] > 300_000:  # 5 min in ms
            # Claim and reprocess
            r.xclaim("email-jobs", "workers", consumer_name,
                     300_000, [entry["message_id"]])
```

Redis Queue Comparison

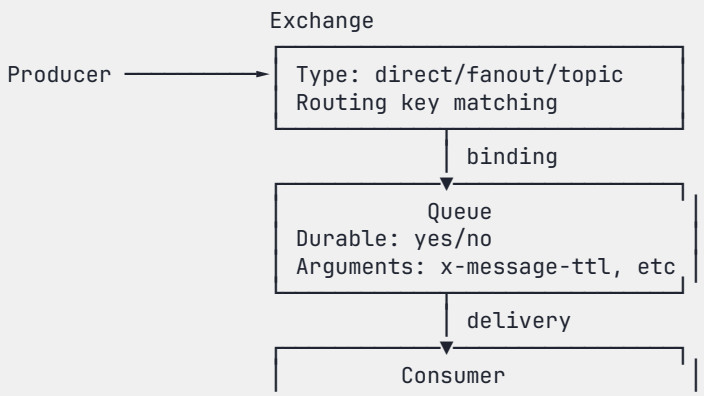
Method	Reliability	Throughput	Complexity	Use When
LPUSH+BRPOP	Low	500K/sec	Minimal	Dev/testing only
BRPOPLPUSH	Medium	400K/sec	Low	< 10K msgs/sec, simple
Redis Streams	High	400K/sec	Medium	Production, need ACK

Cảnh báo

```
BRPOP-based queue: message mất khi worker crash.
Sequence:
  1. BRPOP → message dequeued (removed from Redis)
  2. Worker crashes (before processing)
  3. Message GONE — không cách nào recover

Solution: luôn dùng BRPOPLPUSH hoặc Redis Streams trong production.

Redis persistence:
  - Default (no persistence): mất toàn bộ queue khi Redis restart
  - AOF (appendonly yes): mất tối đa 1 second của writes
  - RDB: mất tối đa N phút (snapshot interval)

Nếu queue data quan trọng (payments, orders): dùng PostgreSQL queue
hoặc RabbitMQ với durable queues + publisher confirms.
```

RabbitMQ

Core Concepts



```
                    │ Prefetch: 1-100          │
                    │ Ack: manual/auto         │
                    └──────────────────────────┘
```

Exchange Types

```python
import pika
import json

connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

# 1. DIRECT Exchange — route by exact routing key
channel.exchange_declare('notifications', exchange_type='direct', durable=True)
channel.queue_declare('email-queue', durable=True)
channel.queue_bind('email-queue', 'notifications', routing_key='email')
channel.queue_bind('sms-queue', 'notifications', routing_key='sms')

# Publish email notification
channel.basic_publish(
    exchange='notifications',
    routing_key='email',        # → goes to email-queue only
    body=json.dumps({"user_id": 123}),
    properties=pika.BasicProperties(
        delivery_mode=2,        # persistent (survive broker restart)
        content_type='application/json'
    )
)

# 2. FANOUT Exchange — broadcast to ALL bound queues
channel.exchange_declare('events', exchange_type='fanout', durable=True)
# All queues bound to 'events' receive every message
# Use case: event broadcasting (order created → notify inventory + email + analytics)

# 3. TOPIC Exchange — pattern matching routing key
channel.exchange_declare('logs', exchange_type='topic', durable=True)
channel.queue_bind('critical-alerts', 'logs', routing_key='*.error.*')    # any error
channel.queue_bind('payment-logs', 'logs', routing_key='payment.#')       # all payment events
channel.queue_bind('all-logs', 'logs', routing_key='#')                   # everything
```

Consumer với Manual Ack + Prefetch

```python
def process_message(ch, method, properties, body):
    try:
        job = json.loads(body)
        do_work(job)
        # Manual ack: message removed from queue only after success
        ch.basic_ack(delivery_tag=method.delivery_tag)
    except TemporaryError:
        # Nack + requeue: message goes back to queue
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
    except PermanentError:
        # Nack without requeue: message goes to DLX (dead letter exchange)
        ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)

# Prefetch: chỉ nhận 1 unacked message tại một lúc
# Prevents: fast consumer hoarding messages, slow consumers starving
channel.basic_qos(prefetch_count=1)   # safest default
# Tune up when:
#   - Processing is fast (< 100ms)
#   - Workers are homogeneous (same speed)
#   - Network round-trip is significant
channel.basic_qos(prefetch_count=10)  # better throughput if workers are reliable

channel.basic_consume('email-queue', process_message)
channel.start_consuming()
```

Durable Queues + Publisher Confirms

```
# Durability stack: cần cả 3 để đảm bảo no message loss

# 1. Durable queue (survive broker restart)
channel.queue_declare('critical-jobs', durable=True)

# 2. Persistent messages (written to disk)
properties = pika.BasicProperties(delivery_mode=2)  # 1=transient, 2=persistent

# 3. Publisher confirms (broker acks the broker received message)
channel.confirm_delivery()  # enable confirm mode

try:
    channel.basic_publish(
        exchange='',
        routing_key='critical-jobs',
        body=json.dumps(job),
        properties=properties,
        mandatory=True      # raise if no queue can receive message
    )
    # Nếu không raise → broker đã nhận và persist
except pika.exceptions.UnroutableError:
    # Không có queue nào nhận — handle tại đây
    log_unroutable(job)
```

Dead Letter Exchange (DLX)

```
# Setup DLX: messages rejected/expired/max-retried → DLX
channel.exchange_declare('dlx', exchange_type='direct', durable=True)
channel.queue_declare('failed-jobs', durable=True)
channel.queue_bind('failed-jobs', 'dlx', routing_key='failed')

# Main queue với DLX config
channel.queue_declare(
    'jobs',
    durable=True,
    arguments={
        'x-dead-letter-exchange': 'dlx',
        'x-dead-letter-routing-key': 'failed',
        'x-message-ttl': 300_000,      # 5 min TTL — expired → DLX
        'x-max-retries': 3             # không standard, implement trong consumer
    }
)

# Retry logic trong consumer
def process_with_retry(ch, method, properties, body):
    headers = properties.headers or {}
    retry_count = headers.get('x-retry-count', 0)

    try:
        do_work(json.loads(body))
        ch.basic_ack(delivery_tag=method.delivery_tag)
    except Exception as e:
        if retry_count < 3:
            # Re-publish với incremented retry count
            new_headers = {**headers, 'x-retry-count': retry_count + 1}
            # Add delay: 10s, 30s, 60s (implement via separate delay queue)
            delay = [10_000, 30_000, 60_000][retry_count]
            republish_with_delay(body, new_headers, delay)
            ch.basic_ack(delivery_tag=method.delivery_tag)  # ack original
        else:
            # Max retries → let it go to DLX via nack
            ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
```

Cảnh báo

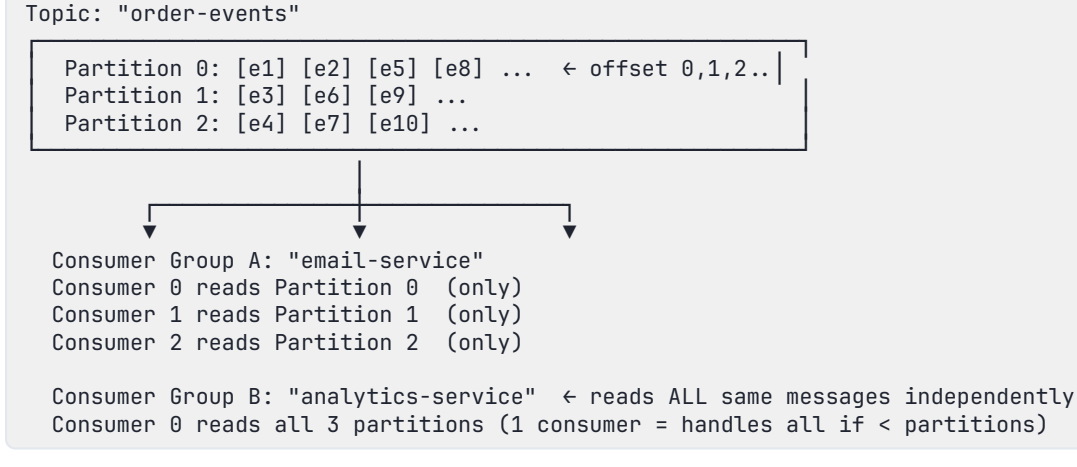
```
Classic mirrored queues deprecated since RabbitMQ 3.13.
Dùng Quorum Queues cho HA:

# Quorum queue: Raft-based replication, majority must ack
channel.queue_declare(
    'critical-jobs',
    durable=True,
    arguments={'x-queue-type': 'quorum'}
)

Quorum vs Classic Mirrored:
  - Quorum: Raft consensus, no split-brain, predictable failover (< 1s)
  - Mirrored: async replication, can lose messages during failover
  - Quorum: requires 3+ nodes (odd number for quorum)
  - Quorum: slightly higher latency (consensus overhead)

Lazy queues (cho large backlogs):
channel.queue_declare(
    'batch-jobs',
    durable=True,
    arguments={'x-queue-mode': 'lazy'}  # messages go to disk immediately
)
# Reduces RAM usage 10-50x for large queues
# Tradeoff: higher latency (disk I/O on enqueue)
```

Kafka — Event Streaming

Kafka vs RabbitMQ — Khi nào dùng cái nào?

```
RabbitMQ                          Kafka
──────────────────────────────── ──────────────────────────────────
Task queue (1 consumer per msg)  Event log (many consumers per msg)
Messages deleted after consume   Messages retained N days/forever
Push-based delivery              Pull-based (consumer controls pace)
< 100K msg/sec typical           1M+ msg/sec per partition
Complex routing (exchanges)      Simple routing (topics + partitions)
Built-in DLQ, retry, TTL         Manual DLQ, retry logic
3-5 nodes typical                6-12+ nodes for HA
Kafka overkill for simple tasks  Overkill for task queues

Use RabbitMQ: background jobs, email/SMS, simple task processing
Use Kafka: event sourcing, CDC, audit logs, analytics pipelines, replay
```

Core Architecture



Producer — Publish Events

```
from confluent_kafka import Producer
import json

producer = Producer({
    'bootstrap.servers': 'kafka1:9092,kafka2:9092,kafka3:9092',
    'acks': 'all',              # wait for all ISR replicas to ack
    'retries': 3,
    'linger.ms': 5,             # batch messages for 5ms before send
    'batch.size': 65536,        # 64KB batch size
    'compression.type': 'zstd'  # compress batches
})

def delivery_report(err, msg):
    if err is not None:
        log_error(f"Message failed: {err}")
    else:
        log_info(f"Delivered to {msg.topic()} [{msg.partition()}] @ offset {msg.offset()}")

# Publish order event
def publish_order_created(order: dict):
    producer.produce(
        topic='order-events',
        key=str(order['id']).encode(),      # same key → same partition (ordering)
        value=json.dumps(order).encode(),
        callback=delivery_report
    )
    producer.poll(0)  # trigger delivery callbacks

# Flush tất cả pending messages trước khi exit
producer.flush(timeout=10)
```

Consumer — With Consumer Groups

```
from confluent_kafka import Consumer, KafkaException

consumer = Consumer({
    'bootstrap.servers': 'kafka1:9092,kafka2:9092',
    'group.id': 'email-service',
    'auto.offset.reset': 'earliest',   # start từ đầu nếu no committed offset
    'enable.auto.commit': False,        # manual commit = safer
    'max.poll.interval.ms': 300_000,   # 5 min processing timeout
})

consumer.subscribe(['order-events'])

try:
    while True:
        msg = consumer.poll(timeout=1.0)

        if msg is None:
            continue
        if msg.error():
            raise KafkaException(msg.error())

        # Process message
        order = json.loads(msg.value())
        send_order_confirmation_email(order)

        # Manual commit: only after successful processing
        consumer.commit(asynchronous=False)  # sync commit = safer, slower

except KeyboardInterrupt:
    pass
finally:
    consumer.close()  # commits offsets, notifies group coordinator
```

Partitioning Strategy — Throughput vs Ordering

```
# Ordering guarantee: ONLY within same partition
# Tradeoff: more partitions = more parallelism BUT more resources

# Case 1: Cần ordering per user (e.g., user activity events)
producer.produce(
    topic='user-events',
    key=str(user_id).encode(),  # same user → same partition → ordered
    value=event_json
)

# Case 2: Cần max throughput (không cần ordering)
producer.produce(
    topic='analytics-events',
    key=None,  # round-robin across partitions → max throughput
    value=event_json
)

# Partition count sizing math:
# Target throughput: 100K msg/sec
# Per-partition throughput: ~10K msg/sec (typical)
# Required partitions: 100K / 10K = 10 partitions minimum
# Add 20% headroom: 12 partitions

# Consumer scaling:
# Max useful consumers in a group = number of partitions
# 10 partitions → max 10 consumers useful
# 11th consumer = idle (no partition assigned)
```

Retention & Compaction

```
# Time-based retention (default 7 days)
kafka-topics.sh --alter --topic order-events \
  --config retention.ms=604800000  # 7 days in ms

# Size-based retention
kafka-topics.sh --alter --topic metrics \
  --config retention.bytes=10737418240  # 10GB per partition

# Log compaction: keep only latest value per key
# Use case: materialized views, cache invalidation, entity state
kafka-topics.sh --alter --topic user-profiles \
  --config cleanup.policy=compact \
  --config min.cleanable.dirty.ratio=0.5

# Combined: compact + delete (keep compacted log, still delete old segments)
--config cleanup.policy=compact,delete
--config retention.ms=2592000000  # 30 days
```

Cảnh báo ⚠

```
Kafka cần ZooKeeper (trước 3.0) hoặc KRaft (từ 3.0, production-ready từ 3.3).

KRaft mode (không cần ZooKeeper):
  KAFKA_PROCESS_ROLES=broker,controller
  KAFKA_NODE_ID=1
  KAFKA_CONTROLLER_QUORUM_VOTERS=1@kafka1:9093,2@kafka2:9093,3@kafka3:9093

Kafka không phải drop-in cho RabbitMQ:
  - Kafka: pull-based, consumer controls pace, low push overhead
  - RabbitMQ: push-based, broker tracks consumer state
  - Kafka for 1M msg/sec; RabbitMQ for complex routing + built-in retry

Ordering only within partition — quan trọng nhất:
  - Order create (partition 0) và order cancel (partition 1) → parallel processing
```

```
  - Cancel có thể process TRƯỚC create → race condition
  - Fix: route all events for same order to same partition (key=order_id)
```

PostgreSQL as Queue ℹ

SKIP LOCKED Pattern

```sql
-- Setup: jobs table
CREATE TABLE jobs (
    id          BIGSERIAL PRIMARY KEY,
    queue       TEXT NOT NULL DEFAULT 'default',
    status      TEXT NOT NULL DEFAULT 'pending',  -- pending/processing/done/failed
    priority    INT NOT NULL DEFAULT 0,
    payload     JSONB NOT NULL,
    attempts    INT NOT NULL DEFAULT 0,
    max_attempts INT NOT NULL DEFAULT 3,
    scheduled_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    locked_at   TIMESTAMPTZ,
    locked_by   TEXT,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

-- Critical indexes
CREATE INDEX idx_jobs_queue_status_priority
    ON jobs (queue, status, priority DESC, scheduled_at ASC)
    WHERE status = 'pending';                          -- partial index, much smaller
```

```sql
-- Worker: fetch + lock in 1 atomic operation
-- SKIP LOCKED: skip rows locked by other workers (no waiting, no deadlock)
WITH next_job AS (
    SELECT id FROM jobs
    WHERE queue = 'default'
      AND status = 'pending'
      AND scheduled_at ⩽ NOW()
    ORDER BY priority DESC, scheduled_at ASC
    LIMIT 1
    FOR UPDATE SKIP LOCKED    -- key: skip instead of wait
)
UPDATE jobs
SET status = 'processing',
    locked_at = NOW(),
    locked_by = 'worker-1',
    attempts = attempts + 1
FROM next_job
WHERE jobs.id = next_job.id
RETURNING jobs.*;
```

```python
import asyncpg
import asyncio

async def fetch_job(conn, queue='default'):
    return await conn.fetchrow("""
        WITH next_job AS (
            SELECT id FROM jobs
            WHERE queue = $1
              AND status = 'pending'
              AND scheduled_at ⩽ NOW()
            ORDER BY priority DESC, scheduled_at ASC
            LIMIT 1
            FOR UPDATE SKIP LOCKED
        )
        UPDATE jobs
        SET status = 'processing', locked_at = NOW(), locked_by = $2, attempts = attempts + 1
        FROM next_job
```

```
        WHERE jobs.id = next_job.id
        RETURNING jobs.*
    """, queue, 'worker-1')

async def complete_job(conn, job_id: int):
    await conn.execute("UPDATE jobs SET status = 'done' WHERE id = $1", job_id)

async def fail_job(conn, job_id: int, error: str, max_attempts: int = 3):
    # Tự động retry nếu chưa đạt max attempts
    await conn.execute("""
        UPDATE jobs
        SET status = CASE WHEN attempts >= max_attempts THEN 'failed' ELSE 'pending' END,
            scheduled_at = NOW() + INTERVAL '1 minute' * attempts,  -- exponential delay
            error_message = $2
        WHERE id = $1
    """, job_id, error)

async def worker_loop(pool):
    async with pool.acquire() as conn:
        while True:
            job = await fetch_job(conn)
            if not job:
                await asyncio.sleep(1)  # polling interval
                continue

            try:
                await process_job(dict(job))
                await complete_job(conn, job['id'])
            except Exception as e:
                await fail_job(conn, job['id'], str(e))
```

Transactional Advantage

```
# Killer feature của PG queue: job + side effects trong 1 transaction
async def create_order_with_jobs(pool, order_data: dict):
    async with pool.acquire() as conn:
        async with conn.transaction():
            # 1. Create order
            order = await conn.fetchrow(
                "INSERT INTO orders (user_id, total) VALUES ($1, $2) RETURNING *",
                order_data['user_id'], order_data['total']
            )

            # 2. Enqueue jobs — trong cùng transaction
            await conn.execute(
                "INSERT INTO jobs (queue, payload) VALUES ('email', $1)",
                json.dumps({"order_id": order['id'], "template": "confirmation"})
            )
            await conn.execute(
                "INSERT INTO jobs (queue, payload) VALUES ('inventory', $1)",
                json.dumps({"order_id": order['id'], "items": order_data['items']})
            )

            # Nếu transaction fail → order VÀ jobs đều rollback
            # Không bao giờ có order mà không có jobs (hoặc ngược lại)
            # Giải quyết hoàn toàn dual-write problem
```

Cảnh báo ❗

```
PG queue max throughput: ~1000-5000 jobs/sec (tuned).
Beyond that: connection overhead + MVCC bloat + autovacuum pressure.

Signs you've outgrown PG queue:
  - Jobs table > 1M rows (even with cleanup)
  - Workers waiting > 100ms for SKIP LOCKED
  - Autovacuum running constantly on jobs table
  - Worker CPU > 50% just on DB queries
```

```
Migration path: PG queue → Redis Streams → RabbitMQ → Kafka

QUAN TRỌNG — Index phải tồn tại:
  - Không có index: SKIP LOCKED scan toàn bộ table
  - Jobs table 1M rows, không index: 500ms+ per fetch
  - Với partial index trên status='pending': < 1ms per fetch

Cleanup: DELETE completed/failed jobs regularly
  DELETE FROM jobs WHERE status IN ('done', 'failed') AND created_at < NOW() - INTERVAL '7 days';
  -- Chạy trong off-peak hoặc dùng pg_partman cho time-partitioned jobs table
```

Event-Driven Architecture A

Events vs Commands

```
Command (Intent — future action):
  "CreateOrder" — yêu cầu làm gì đó, 1 handler, expect response

Event (Fact — past occurrence):
  "OrderCreated" — điều gì đó đã xảy ra, nhiều handlers, fire-and-forget

Command flow:
  POST /orders → OrderService.CreateOrder() → returns created order

Event flow:
  OrderService publishes "OrderCreated"
  EmailService subscribes → sends confirmation
  InventoryService subscribes → decrements stock
  AnalyticsService subscribes → updates metrics
  (tất cả độc lập, không biết nhau)
```

Outbox Pattern — Giải quyết Dual-Write A

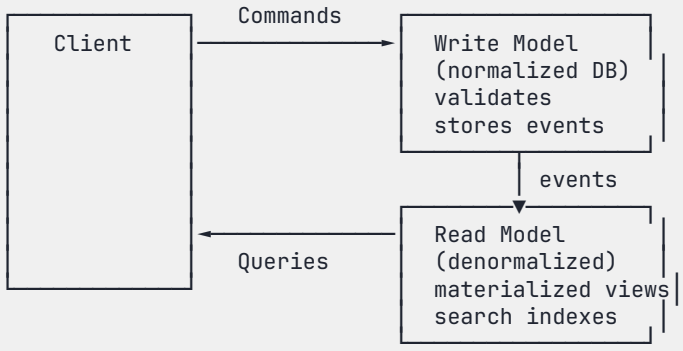
```
Vấn đề dual-write:
  1. Write to DB ✓
  2. Publish to Kafka × (network fail)
  → DB và Kafka inconsistent

Outbox pattern:
  1. Write to DB + write to outbox table (same transaction) ✓
  2. Separate process reads outbox → publishes to Kafka
  → At-least-once delivery, no inconsistency
```

```sql
-- Outbox table
CREATE TABLE outbox_events (
    id          BIGSERIAL PRIMARY KEY,
    aggregate_type TEXT NOT NULL,   -- "Order", "User"
    aggregate_id   TEXT NOT NULL,
    event_type     TEXT NOT NULL,   -- "OrderCreated"
    payload        JSONB NOT NULL,
    created_at     TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    published_at   TIMESTAMPTZ,     -- NULL = not yet published
    published      BOOLEAN NOT NULL DEFAULT FALSE
);
```

```python
async def create_order(conn, order_data: dict):
    async with conn.transaction():
        # 1. Business logic
        order = await conn.fetchrow(
            "INSERT INTO orders (user_id, total) VALUES ($1, $2) RETURNING *",
            order_data['user_id'], order_data['total']
        )
```

```
        # 2. Write to outbox (same transaction = atomic)
        await conn.execute("""
            INSERT INTO outbox_events (aggregate_type, aggregate_id, event_type, payload)
            VALUES ('Order', $1, 'OrderCreated', $2)
        """, str(order['id']), json.dumps({
            "order_id": order['id'],
            "user_id": order['user_id'],
            "total": str(order['total'])
        }))

        return order
        # Transaction commits → both records saved or neither

# Outbox publisher (separate process, runs continuously)
async def publish_outbox_events(conn, kafka_producer):
    while True:
        # Fetch unpublished events
        events = await conn.fetch("""
            SELECT * FROM outbox_events
            WHERE published = FALSE
            ORDER BY id ASC
            LIMIT 100
            FOR UPDATE SKIP LOCKED
        """)

        for event in events:
            kafka_producer.produce(
                topic=event['aggregate_type'].lower() + '-events',
                key=event['aggregate_id'].encode(),
                value=event['payload'].encode()
            )

        kafka_producer.flush()

        # Mark as published
        await conn.execute("""
            UPDATE outbox_events
            SET published = TRUE, published_at = NOW()
            WHERE id = ANY($1)
        """, [e['id'] for e in events])

        await asyncio.sleep(0.1)  # 100ms polling
```

CQRS — Command Query Responsibility Segregation A




```
Write model: consistent, normalized, slow reads OK
Read model: eventual consistent, optimized for queries, fast reads
```

```
# Write side: Command handler
class OrderCommandHandler:
    async def handle_create_order(self, cmd: CreateOrderCommand):
        # Validate
        if not await self.inventory.check_availability(cmd.items):
            raise InsufficientInventoryError()
```

```
        # Create domain event
        event = OrderCreatedEvent(
            order_id=uuid4(),
            user_id=cmd.user_id,
            items=cmd.items,
            total=cmd.total,
            timestamp=datetime.utcnow()
        )

        # Store event (event store)
        await self.event_store.append('order', event.order_id, event)
        # Publish for read model projection
        await self.event_bus.publish(event)

# Read side: Event handler updates denormalized read model
class OrderReadModelProjector:
    async def on_order_created(self, event: OrderCreatedEvent):
        # Update denormalized table (fast reads, no joins)
        await self.read_db.execute("""
            INSERT INTO orders_view
            (order_id, user_name, user_email, item_names, total, status, created_at)
            VALUES ($1, $2, $3, $4, $5, $6, $7)
        """,
        event.order_id, event.user.name, event.user.email,
        [i.name for i in event.items], event.total, 'pending', event.timestamp)
```

Saga Pattern — Distributed Transactions A

```
Distributed transaction: tạo Order → deduct Inventory → charge Payment
3 services, không có global transaction manager

Choreography Saga (event-driven):
  OrderService: OrderCreated →
  InventoryService: InventoryReserved →
  PaymentService: PaymentCharged →
  OrderService: OrderCompleted

Compensation (rollback) nếu Payment fail:
  PaymentService: PaymentFailed →
  InventoryService: InventoryReleased (compensating transaction) →
  OrderService: OrderCancelled (compensating transaction)

Orchestration Saga (coordinator):
  SagaOrchestrator sends Commands, receives Events
  Orchestrator tracks saga state
  Orchestrator triggers compensating transactions on failure
```

Task Scheduling I

Distributed Cron — Exactly-Once Execution

```
# Problem: 3 instances của app, tất cả có cron job chạy lúc 2AM
# Result: job chạy 3 lần → duplicate processing

# Solution: advisory lock trước khi execute
import asyncpg

async def run_exclusive_job(pool, job_name: str, job_fn):
    async with pool.acquire() as conn:
        # pg_try_advisory_lock: non-blocking, returns False nếu lock held
        # Lock tự động release khi connection đóng (end of with block)
        acquired = await conn.fetchval(
            "SELECT pg_try_advisory_lock(hashtext($1))", job_name
        )
```

```
        if not acquired:
            return  # Another instance is running this job

        try:
            await job_fn()
        finally:
            await conn.execute(
                "SELECT pg_advisory_unlock(hashtext($1))", job_name
            )

# Usage với APScheduler
from apscheduler.schedulers.asyncio import AsyncIOScheduler

scheduler = AsyncIOScheduler()

@scheduler.scheduled_job('cron', hour=2, minute=0)
async def daily_report():
    await run_exclusive_job(pool, 'daily_report', generate_and_send_report)

scheduler.start()
```

```
# pg_cron: Cron jobs chạy trong PostgreSQL (single node, no duplication)
CREATE EXTENSION pg_cron;

-- Chạy cleanup mỗi đêm 3AM
SELECT cron.schedule('cleanup-old-jobs', '0 3 * * *',
    'DELETE FROM jobs WHERE status = ''done'' AND created_at < NOW() - INTERVAL ''7 days''');

-- Refresh materialized view mỗi 5 phút
SELECT cron.schedule('refresh-stats', '*/5 * * * *',
    'REFRESH MATERIALIZED VIEW CONCURRENTLY order_stats');

-- List scheduled jobs
SELECT * FROM cron.job;

-- Unschedule
SELECT cron.unschedule('cleanup-old-jobs');
```

Kubernetes CronJob — Cảnh báo ❗

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: daily-report
spec:
  schedule: "0 2 * * *"
  concurrencyPolicy: Forbid   # QUAN TRỌNG: không cho phép concurrent runs
                              # Allow (default): nhiều instances chạy đồng thời!
                              # Forbid: skip nếu previous job chưa xong
                              # Replace: kill previous, start new
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  startingDeadlineSeconds: 300  # nếu missed schedule, chỉ retry trong 5 min
  jobTemplate:
    spec:
      backoffLimit: 2          # retry 2 lần nếu fail
      activeDeadlineSeconds: 3600  # kill job nếu chạy > 1 hour
      template:
        spec:
          restartPolicy: OnFailure
          containers:
          - name: report-job
            image: myapp:latest
            command: ["python", "generate_report.py"]
            resources:
              requests:
                cpu: "500m"
                memory: "256Mi"
```

```
            limits:
              cpu: "2"
              memory: "16Gi"
```

Webhook & Callback Patterns ℹ

Reliable Webhook Delivery

```python
# Webhook retry với exponential backoff
import httpx
import asyncio
import hmac, hashlib, json

async def deliver_webhook(
    endpoint: str,
    payload: dict,
    secret: str,
    max_retries: int = 5
):
    body = json.dumps(payload).encode()

    # HMAC signature: receiver can verify payload authenticity
    signature = hmac.new(
        secret.encode(),
        body,
        hashlib.sha256
    ).hexdigest()

    headers = {
        "Content-Type": "application/json",
        "X-Webhook-Signature": f"sha256={signature}",
        "X-Webhook-Id": payload.get("event_id"),      # idempotency key
        "X-Webhook-Timestamp": str(int(time.time())),
    }

    for attempt in range(max_retries):
        try:
            async with httpx.AsyncClient(timeout=10) as client:
                response = await client.post(endpoint, content=body, headers=headers)

            if response.status_code in (200, 201, 202, 204):
                return  # Success

            # 4xx (except 429): don't retry (client error)
            if 400 ⩽ response.status_code < 500 and response.status_code =≠ 429:
                log_webhook_error(endpoint, response.status_code, attempt)
                return

        except (httpx.TimeoutException, httpx.ConnectError):
            pass  # Will retry

        if attempt < max_retries - 1:
            # Exponential backoff: 1s, 2s, 4s, 8s, 16s
            delay = 2 ** attempt
            await asyncio.sleep(delay)

    # Max retries exceeded → dead letter
    await save_to_dlq(endpoint, payload)
```

```python
# Receiver: verify signature + idempotency
@app.post("/webhooks/orders")
async def receive_order_webhook(request: Request):
    body = await request.body()

    # Verify signature
```

```
    signature = request.headers.get("X-Webhook-Signature", "")
    expected = "sha256=" + hmac.new(
        WEBHOOK_SECRET.encode(), body, hashlib.sha256
    ).hexdigest()

    if not hmac.compare_digest(signature, expected):
        raise HTTPException(status_code=401, detail="Invalid signature")

    # Idempotency: skip if already processed
    webhook_id = request.headers.get("X-Webhook-Id")
    if await redis.exists(f"webhook:{webhook_id}"):
        return {"status": "already_processed"}

    # Process
    event = json.loads(body)
    await process_order_event(event)

    # Mark as processed (expire after 24h)
    await redis.setex(f"webhook:{webhook_id}", 86400, "1")

    return {"status": "ok"}
```

Tổng kết — Chọn Queue đúng cho Use Case

```
Use Case                          Solution              Why
──────────────────────────────────────────────────────────────────────
Simple background jobs            Redis Streams         Fast, low infra
Email/SMS/Push notifications       RabbitMQ              DLX, retry, TTL built-in
High throughput events (1M+)      Kafka                 Partitioned, scalable
Event sourcing / audit log        Kafka                 Retention, replay
Transactional jobs                PostgreSQL SKIP LOCKED Atomic with business data
Distributed transaction           Saga (Kafka/RabbitMQ) Compensation pattern
Real-time notifications           SSE + Redis Pub/Sub   Simple, HTTP-native
Chat / collaboration              WebSocket + Redis     Bidirectional, low latency
Scheduled tasks (simple)          pg_cron               No extra infra
Scheduled tasks (distributed)     K8s CronJob + advisory lock  HA, exactly-once

Golden rule:
  Start with PG queue (zero infra, transactional)
  Graduate to Redis Streams (speed, no extra server)
  Graduate to RabbitMQ (complex routing, DLQ, HA)
  Graduate to Kafka (1M+ msg/sec, event sourcing, CDC)
  Don't jump to Kafka prematurely — it's powerful and expensive to operate.
```

> Queue không phải magic — nó chỉ chuyển failure từ "ngay lập tức" sang "sau đó". Thiết kế idempotent jobs, monitor queue depth, và có alerting khi backlog > threshold.

23

Load Balancing & Reverse Proxy — Distribute 1M connections across small servers

> **Tags:** I Intermediate — A Advanced
> **Series:** optimization-08 / 08
> **Liên quan:** optimization-02 (kernel tuning), optimization-03 (connection pooling), optimization-04 (caching)

Tại sao cần Load Balancing?

Một VPS $20/tháng có thể handle ~5,000 concurrent connections sau khi kernel đã tuned. Để đạt 1 triệu connections, bạn cần phân tải sang nhiều server. Load balancer là lớp "bộ phối mạng" nhận toàn bộ traffic rồi phân ra backend pool.

```
Internet → [Load Balancer] → [App Server 1]
                           → [App Server 2]
                           → [App Server 3]
                           → [App Server N]
```

Lợi ích chính: - **High availability**: một backend chết, traffic tự động chuyển sang server còn lại - **Horizontal scaling**: thêm server mới mà không cần downtime - **SSL offload**: LB xử lý TLS, backend chạy plain HTTP → tiết kiệm CPU backend - **Traffic control**: rate limiting, health checks, request routing theo URL/header

L4 vs L7 Load Balancing I

L4 — Transport Layer (TCP/UDP)

Route traffic theo IP:port, không đọc nội dung gói tin. Nhanh nhất vì không parse HTTP.

```
Client → LB:443 → Backend:8080  (chỉ forward TCP stream, không hiểu HTTP)
```

Khi dùng L4: - Plain TCP services: databases, game servers, custom protocols - Non-HTTP: SMTP, MQTT, gRPC thuần TCP - Maximum throughput cần — L4 throughput gấp 2-5x L7 cùng phần cứng - Database load balancing (MySQL, PostgreSQL, Redis)

Tools L4: - HAProxy `mode tcp` - Nginx `stream {}` block - iptables DNAT (kernel-level, zero userspace overhead) - AWS NLB, GCP Network LB

```
# Nginx L4 — forward PostgreSQL traffic
stream {
    upstream postgres_backends {
        server 10.0.1.10:5432;
```

```
        server 10.0.1.11:5432;
    }
    server {
        listen 5432;
        proxy_pass postgres_backends;
        proxy_timeout 3600s;
        proxy_connect_timeout 5s;
    }
}
```

L7 — Application Layer (HTTP/HTTPS)

Đọc HTTP headers, URL path, cookies để đưa ra quyết định routing thông minh.

```
Client → LB (đọc: GET /api/users) → Backend API
       → LB (đọc: GET /static/img) → CDN/Static server
```

Khi dùng L7: - URL-based routing: `/api/*` → API cluster, `/admin/*` → admin cluster - SSL/TLS termination tại LB - Header manipulation: thêm `X-Real-IP`, `X-Forwarded-For` - Response caching - Rate limiting theo User-Agent, path, IP

Hybrid pattern (large scale):

```
Internet → [L4 LB cluster] → [L7 LB cluster] → [Backend cluster]
           (iptables/ECMP)    (Nginx/HAProxy)    (App servers)
```

L4 phân traffic vào pool L7 nodes → L7 làm routing thông minh. Phổ biến ở scale triệu+ users.

Nginx as Load Balancer ℹ️

Cấu hình cơ bản

```
http {
    # Định nghĩa pool backend
    upstream app_backend {
        # Thuật toán: round-robin (default) — request luân phiên qua các server
        # least_conn — gửi tới server ít connection nhất (tốt cho long-lived connections)
        # ip_hash — cùng IP luôn vào cùng backend (sticky sessions)
        # random two — chọn ngẫu nhiên 2 server, pick server ít conn hơn (Power of Two Choices)
        least_conn;

        server 10.0.1.10:8080 weight=3;  # server mạnh hơn nhận 3x traffic
        server 10.0.1.11:8080 weight=2;
        server 10.0.1.12:8080 weight=1;

        # Health check (passive): đánh dấu server fail sau 3 lần lỗi trong 30s
        # server tự recover sau 30s nếu request tiếp theo thành công
        server 10.0.1.13:8080 max_fails=3 fail_timeout=30s;

        # Backup: chỉ nhận traffic khi tất cả server chính fail
        server 10.0.1.14:8080 backup;

        # Upstream keepalive: tái dùng TCP connections đến backend
        # ❗ QUAN TRỌNG: không có keepalive = tạo TCP connection mới mỗi request
        # 10K req/s × 3ms TCP handshake = 30s CPU wasted mỗi giây
        keepalive 100;          # giữ tối đa 100 idle connections trong pool
        keepalive_timeout 60s;  # đóng idle connection sau 60s
        keepalive_requests 1000; # sau 1000 requests, đóng và tạo connection mới
    }

    server {
        listen 80;
```

```
        server_name example.com;

        location / {
            proxy_pass http://app_backend;

            # Bắt buộc khi dùng upstream keepalive
            proxy_http_version 1.1;
            proxy_set_header Connection "";  # xóa Connection: close

            # Forward IP thật của client
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header Host $host;

            # Timeout settings
            proxy_connect_timeout 5s;   # timeout kết nối tới backend
            proxy_send_timeout 60s;     # timeout gửi request
            proxy_read_timeout 60s;     # timeout đợi response
        }
    }
}
```

Worker tuning cho 1M connections A

```
# /etc/nginx/nginx.conf

# = số CPU cores. 4 cores × 65535 connections = 262K max.
# Để đạt 1M: cần nhiều LB nodes hoặc nhiều cores hơn.
worker_processes auto;

# Giới hạn file descriptors của process này
# Phải match với system limit: ulimit -n
worker_rlimit_nofile 200000;

events {
    # Số connections tối đa mỗi worker
    worker_connections 65535;

    # epoll = Linux async I/O, hiệu quả nhất cho C10K+
    use epoll;

    # Chấp nhận nhiều connections cùng lúc thay vì từng cái
    # ! Tắt nếu có nhiều workers cạnh tranh gây thundering herd
    multi_accept on;
}

http {
    # Tắt sendfile nếu dùng proxy (không serve file trực tiếp)
    sendfile on;
    tcp_nopush on;      # gom packets, giảm số lần gọi send()
    tcp_nodelay on;     # với keepalive connections, gửi ngay không đợi buffer

    # Keepalive với client
    keepalive_timeout 75s;
    keepalive_requests 10000;  # sau 10K requests, client phải reconnect
}
```

Buffering và Caching I

```
location /api/ {
    proxy_pass http://app_backend;

    # Buffering ON (default): Nginx buffer toàn bộ response trước khi gửi client
    # → Backend connection được giải phóng ngay sau khi response nhận xong
    # → Client download chậm không làm chậm backend
    proxy_buffering on;
    proxy_buffer_size 16k;      # buffer cho response headers
    proxy_buffers 4 32k;        # 4 buffers × 32k = 128k total buffer
```

```
    proxy_busy_buffers_size 64k;

    # ❗ TẮT buffering cho streaming/SSE/WebSocket
    # proxy_buffering off;
}

# Proxy cache — serve cached response, backend không bị hit
proxy_cache_path /var/cache/nginx/api levels=1:2 keys_zone=api_cache:100m
                 max_size=10g inactive=60m use_temp_path=off;

location /api/public/ {
    proxy_pass http://app_backend;
    proxy_cache api_cache;
    proxy_cache_valid 200 10m;   # cache 200 OK response trong 10 phút
    proxy_cache_valid 404 1m;
    proxy_cache_use_stale error timeout updating; # serve stale nếu backend lỗi
    add_header X-Cache-Status $upstream_cache_status; # HIT/MISS/BYPASS
}
```

Rate Limiting I

```
http {
    # Zone 10MB = track ~160K địa chỉ IP
    # rate=100r/s = tối đa 100 requests/giây mỗi IP
    limit_req_zone $binary_remote_addr zone=api_limit:10m rate=100r/s;

    # Rate limit nghiêm hơn cho login endpoint
    limit_req_zone $binary_remote_addr zone=login_limit:10m rate=5r/m;

    server {
        location /api/ {
            # burst=200: cho phép burst tới 200 req, xử lý ngay (nodelay)
            # Sau khi hết burst, request tiếp theo bị delay hoặc reject (503)
            limit_req zone=api_limit burst=200 nodelay;
        }

        location /auth/login {
            limit_req zone=login_limit burst=3 nodelay;
            limit_req_status 429;  # trả 429 Too Many Requests thay vì 503
        }
    }
}
```

HAProxy I A

HAProxy là load balancer chuyên dụng, hiệu suất cao hơn Nginx ở vai trò LB thuần túy.

Cấu trúc config

```
# /etc/haproxy/haproxy.cfg

global
    maxconn 100000          # tổng max connections toàn bộ HAProxy process
    nbthread 4              # số threads (HAProxy 2.0+). Mặc định = số CPU cores.
    # ❗ Trước HAProxy 2.0: dùng nbproc (deprecated). Upgrade lên 2.x.

    # Unix socket để runtime API (live changes không restart)
    stats socket /run/haproxy/admin.sock mode 660 level admin expose-fd listeners

    log /dev/log local0 info

defaults
    mode http               # http hoặc tcp
    timeout connect 5s
```

```
    timeout client 30s
    timeout server 30s
    timeout queue 30s       # timeout khi connections queue (tất cả backend slots full)
    log global
    option httplog
    option dontlognull      # không log health check requests

#---------- Frontend: nhận traffic từ Internet ----------
frontend http_in
    bind *:80
    bind *:443 ssl crt /etc/ssl/certs/example.pem alpn h2,http/1.1

    # ACL — điều kiện routing
    acl is_api path_beg /api/
    acl is_static path_beg /static/ /assets/ /images/
    acl is_admin path_beg /admin/

    # Routing theo ACL
    use_backend api_servers   if is_api
    use_backend static_servers if is_static
    use_backend admin_servers  if is_admin
    default_backend app_servers

#---------- Backend pools ----------
backend app_servers
    balance leastconn        # least connections — tốt cho long-lived requests
    option httpchk GET /health  # health check endpoint
    http-check expect status 200

    server app1 10.0.1.10:8080 check maxconn 500 inter 2s rise 2 fall 3
    server app2 10.0.1.11:8080 check maxconn 500 inter 2s rise 2 fall 3
    server app3 10.0.1.12:8080 check maxconn 500 inter 2s rise 2 fall 3
    # rise 2: cần 2 health check pass liên tiếp để đưa server trở lại
    # fall 3: cần 3 fail liên tiếp để đánh dấu server down

backend api_servers
    balance roundrobin
    option httpchk GET /api/health
    timeout server 120s      # API có thể slow — tăng timeout riêng

    server api1 10.0.1.20:8080 check
    server api2 10.0.1.21:8080 check

#---------- Stats Dashboard ----------
listen stats
    bind *:8404
    stats enable
    stats uri /stats
    stats refresh 30s
    stats auth admin:secretpassword  # basic auth cho dashboard
```

Stick Tables — Rate Limiting & DDoS Mitigation A

Stick tables lưu trạng thái client trong RAM. Không cần Redis hay external dependency.

```
frontend http_in
    bind *:80

    # Stick table: track connection rate mỗi IP
    # size 1m = 1 triệu entries, expire 30s, lưu conn_rate trong 3 giây
    stick-table type ip size 1m expire 30s store
conn_rate(3s),http_req_rate(10s),http_err_rate(30s)

    # Ghi metric cho mỗi request
    http-request track-sc0 src

    # Reject nếu vượt ngưỡng
    # > 30 connections trong 3s → DDoS/scanner
    http-request deny deny_status 429 if { sc_conn_rate(0) gt 30 }
    # > 100 requests trong 10s → rate limit
```

```
    http-request deny deny_status 429 if { sc_http_req_rate(0) gt 100 }
    # > 20 errors trong 30s → likely bad actor
    http-request deny deny_status 429 if { sc_http_err_rate(0) gt 20 }

    default_backend app_servers
```

HAProxy Runtime API A

Thay đổi cấu hình live, không restart, không drop connection:

```
# Disable một server (maintenance)
echo "disable server app_servers/app1" | socat stdio /run/haproxy/admin.sock

# Enable lại
echo "enable server app_servers/app1" | socat stdio /run/haproxy/admin.sock

# Thay đổi weight của server (shift traffic)
echo "set weight app_servers/app1 50" | socat stdio /run/haproxy/admin.sock

# Xem trạng thái tất cả servers
echo "show servers state" | socat stdio /run/haproxy/admin.sock

# Xem stats (CSV)
echo "show stat" | socat stdio /run/haproxy/admin.sock | cut -d',' -f1,2,18,19

# Hot reload (graceful, không drop connections, kể từ HAProxy 1.8+)
systemctl reload haproxy
```

SSL/TLS Termination I

Tại sao terminate TLS tại LB?

TLS handshake tốn khoảng 1-3ms CPU mỗi connection mới. Với 1M connections/giờ:

```
1,000,000 connections × 2ms = 2,000,000ms = 33 phút CPU/giờ
```

Nếu backend xử lý TLS: 33 phút CPU wasted trên mỗi backend server. Tập trung SSL termination tại LB → backend dùng plain HTTP → tiết kiệm CPU đáng kể.

Nginx SSL config tối ưu

```
server {
    listen 443 ssl http2;
    server_name example.com;

    ssl_certificate /etc/ssl/certs/example.crt;
    ssl_certificate_key /etc/ssl/private/example.key;

    # Chỉ TLS 1.2 và 1.3. TLS 1.3 = 1-RTT handshake (nhanh hơn 1.2 = 2-RTT)
    ssl_protocols TLSv1.2 TLSv1.3;

    # Cipher suites hiện đại
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384;
    ssl_prefer_server_ciphers off;  # TLS 1.3 tự chọn, không cần server preference

    # Session resumption — client kết nối lại không cần full handshake
    # shared:SSL:50m = 50MB shared cache giữa workers = ~200K sessions
    ssl_session_cache shared:SSL:50m;
    ssl_session_timeout 1d;         # session valid trong 1 ngày
    ssl_session_tickets on;         # TLS session tickets (client-side resumption)

    # OCSP Stapling — server đính kèm proof cert còn valid
```

```
    # Không cần client hỏi CA → tiết kiệm 1 roundtrip DNS + HTTP
    ssl_stapling on;
    ssl_stapling_verify on;
    ssl_trusted_certificate /etc/ssl/certs/chain.crt;
    resolver 8.8.8.8 valid=300s;

    # HSTS — báo browser chỉ kết nối HTTPS
    add_header Strict-Transport-Security "max-age=63072000; includeSubDomains; preload";
}
```

DNS Load Balancing ℹ

Round-Robin DNS

Đơn giản nhất: đặt nhiều A record cho cùng một domain.

```
# Cấu hình DNS
example.com.  60  A  203.0.113.10
example.com.  60  A  203.0.113.11
example.com.  60  A  203.0.113.12
```

- Ưu điểm: đơn giản, không cần phần mềm LB riêng
- Nhược điểm: không có health check. Server chết → client vẫn có thể resolve IP chết đó.
- TTL thấp (30-60s) để failover nhanh hơn. Nhưng một số resolver bỏ qua TTL và cache lâu hơn.

Weighted DNS (Route 53 / Cloudflare)

```
# Route 53 weighted routing — 70% traffic vào us-east, 30% vào eu-west
aws route53 change-resource-record-sets --hosted-zone-id Z123 --change-batch '{
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "example.com",
        "Type": "A",
        "SetIdentifier": "us-east",
        "Weight": 70,
        "TTL": 60,
        "ResourceRecords": [{"Value": "203.0.113.10"}]
      }
    },
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "example.com",
        "Type": "A",
        "SetIdentifier": "eu-west",
        "Weight": 30,
        "TTL": 60,
        "ResourceRecords": [{"Value": "198.51.100.20"}]
      }
    }
  ]
}'
```

GeoDNS: route client tới datacenter gần nhất theo vị trí địa lý. Giảm latency đáng kể cho user quốc tế.

CDN as Load Balancer I

CDN (Cloudflare, CloudFront, Fastly) hoạt động như L7 LB toàn cầu với thêm caching và DDoS protection.

```
User (Việt Nam) → Cloudflare edge (Singapore) → Cache HIT → trả ngay
                                               → Cache MISS → Origin server
```

Origin Shield: thay vì mỗi edge node tự fetch từ origin, tất cả cache miss đi qua một "shield" node trung gian. Origin nhận ít request hơn nhiều.

```
Edge nodes (100+ PoP) → Origin Shield (1-2 nodes) → Origin server
100,000 cache misses  → 10,000 consolidated       → Origin xử lý 10K thay vì 100K
```

QUAN TRỌNG: Không có CDN, VPS nhỏ nhận raw traffic từ Internet, bao gồm DDoS. Một cuộc tấn công 10Gbps sẽ làm sập server $20/tháng ngay lập tức. CDN = lớp bảo vệ bắt buộc.

```
// Cloudflare Workers — compute tại edge, giảm tải origin
// Ví dụ: serve A/B test variant tại edge, không cần request tới origin
addEventListener('fetch', event ⇒ {
  event.respondWith(handleRequest(event.request))
})

async function handleRequest(request) {
  const userAgent = request.headers.get('User-Agent')
  // Logic tại edge: không cần round-trip về origin
  if (userAgent.includes('Bot')) {
    return new Response('Not allowed', { status: 403 })
  }
  return fetch(request)  // forward tới origin
}
```

Horizontal Scaling Patterns I A

Stateless Services — nền tảng của horizontal scaling

```
Stateful (BAD):
User session lưu trong RAM của server → chỉ server đó phục vụ được user đó
→ Sticky sessions bắt buộc → không scale ngang được

Stateless (GOOD):
Session lưu trong Redis → mọi server đều xử lý được mọi request
→ LB có thể gửi request tới bất kỳ server nào
→ Thêm server mới → capacity tăng ngay
```

```
# BAD: session trong memory (stateful)
sessions = {}  # biến global trong process

def login(user_id):
    sessions[user_id] = {"logged_in": True, "time": time.time()}

# GOOD: session trong Redis (stateless)
import redis
r = redis.Redis(host='redis-cluster', port=6379)

def login(user_id):
    session_data = json.dumps({"logged_in": True, "time": time.time()})
    r.setex(f"session:{user_id}", 3600, session_data)  # expire sau 1h
```

Sticky Sessions — khi bắt buộc phải dùng I

```
# ip_hash: cùng IP luôn vào cùng backend
# ! Nếu backend chết → user mất session
# ! IP hash không đều nếu user sau NAT → nhiều user vào cùng 1 server
upstream app_backend {
    ip_hash;
    server 10.0.1.10:8080;
    server 10.0.1.11:8080;
}
```

```
# HAProxy cookie-based sticky: chính xác hơn ip_hash
backend app_servers
    balance roundrobin
    cookie SERVERID insert indirect nocache  # HAProxy thêm cookie vào response
    server app1 10.0.1.10:8080 check cookie app1
    server app2 10.0.1.11:8080 check cookie app2
```

Auto-scaling A

```
# Kubernetes HPA — scale theo CPU usage
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: app
  minReplicas: 2
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70  # scale up khi CPU > 70%
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 60   # đợi 60s trước khi scale up tiếp
      policies:
      - type: Pods
        value: 4                       # thêm tối đa 4 pods mỗi lần scale
        periodSeconds: 60
    scaleDown:
      stabilizationWindowSeconds: 300  # đợi 5 phút trước khi scale down (tránh flapping)
```

Benchmark & Monitoring I

```
# Test load balancer throughput với wrk
wrk -t12 -c400 -d30s http://loadbalancer/

# Kết quả mẫu:
# Running 30s test @ http://loadbalancer/
# 12 threads and 400 connections
# Requests/sec: 85,432.17   ← throughput
```

```
# Latency: 4.68ms avg       ← latency
# Transfer/sec: 1.05GB

# Kiểm tra distribution across backends qua HAProxy stats
echo "show stat" | socat stdio /run/haproxy/admin.sock | \
  awk -F',' 'NR>2 {print $1,$2,$48}' | column -t
# Backend     Server  req_rate
# app_servers app1    28405
# app_servers app2    28510
# app_servers app3    28517

# Nginx active connections real-time
watch -n1 'curl -s http://localhost/nginx_status'
# Active connections: 1234
# server accepts handled requests
#  892340 892340 4521809

# Check upstream keepalive effectiveness
curl -s http://localhost/nginx_status | grep "Active connections"
# Số connections nhỏ hơn requests/sec nhiều → keepalive đang hoạt động
```

Checklist triển khai

```
☐ Upstream keepalive bật (proxy_http_version 1.1 + Connection "")
☐ Health check endpoint tồn tại trên backend (/health trả 200)
☐ SSL session cache cấu hình (ssl_session_cache shared:SSL:50m)
☐ TLS 1.3 bật (ssl_protocols TLSv1.2 TLSv1.3)
☐ OCSP stapling bật
☐ Worker connections đủ (worker_connections 65535)
☐ System file descriptors đủ (ulimit -n 200000)
☐ Rate limiting theo IP bật
☐ Upstream backup server cấu hình
☐ CDN hoặc DDoS protection ở phía trước LB
☐ Monitoring: upstream response time, error rate, connection count
☐ Hot reload test: systemctl reload haproxy/nginx không drop connections
```

Xem tiếp: - optimization-02: kernel tuning để tăng max connections - optimization-03: connection pooling giữa LB và backend - optimization-04: caching để giảm tải backend