ADVANCED CACHING

Flawless Application Delivery

Prerequisites/Expectations

- Sysadmin, DevOps, Solution Architect
- Some familiarity with Web Servers
- Some familiarity with Linux
- Text Editor: Vim, Vi, Emacs etc.
- Some knowledge of Networking

The Training Environment

- AWS EC2 Instances
- Ubuntu 16.04
  - Apache 2
  - Wordpress
  - NGINX Plus r11

Log Into AWS

If you haven't done so already, please take the time to SSH into your EC2 Instances (Windows users use PuTTY).

Check your email for the login credentials, check your spam folder!

```
ssh student<number>@<ec2-server-hostname>
```

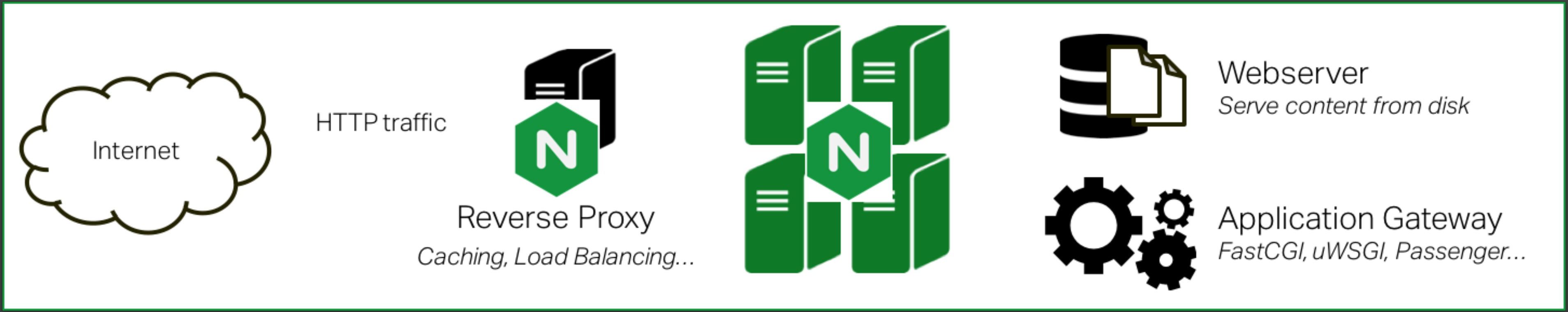
Course Administration

- Course Duration: 4 hours
- Ask questions at any time!

Agenda

- Caching Overview
- Managing Cached Content
- Cache Tuning
- Cache Scaling

NGINX Use Cases



CACHING OVERVIEW

Module Objectives

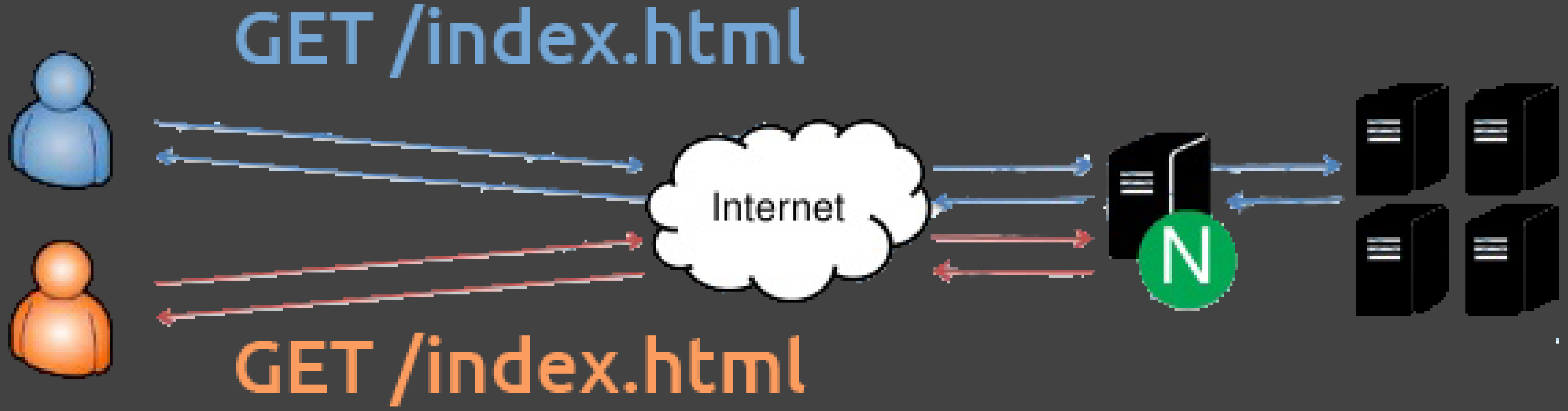
This module enables you to:

- Understand RFC Caching Guidelines
- Review Basic NGINX Cache Configuration
- Debugging the Cache

How Caching Works

Basic Principles

- Browser Cache
- CDN
- Reverse Proxy Cache



HTTP `Cache-Control`

`Cache-Control` header dictates behavior

```
"public"
"private"
"no-cache"
"no-store"
"no-transform"
"must-revalidate"
"proxy-revalidate"
```

Documentation: RFC Guidelines

HTTP Headers

```
Expires: Tue, 6 May 2017 03:18:12 GMT
Cache-Control: public, max-age=60
X-Accel-Expires: 30
Last-Modified: Tue, 29 April 2017 02:28:11 GMT
ETag: "3e74-215-3105fbbc"
```

Documentation: HTTP Header Definitions

X-Accel

```
# When passed URI /protected_files/myfile.pdf
location /protected_files {
  internal;
  alias /var/www/files;
}

# Or proxy to another server
location /protected_files {
  internal;
  proxy_pass http://127.0.0.1:8080;
}
```

Documentation: X-Accel

Special Headers

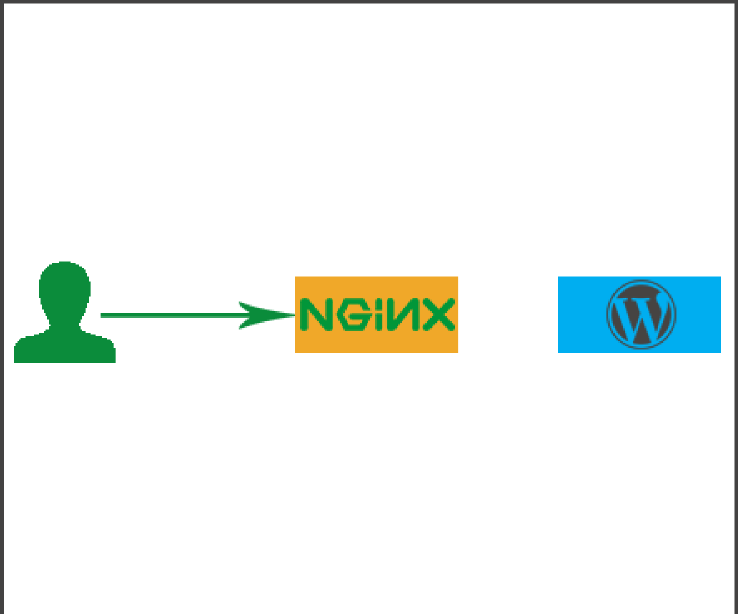
- `X-Accel-Redirect`
- `X-Accel-Buffering`
- `X-Accel-Charset`
- `X-Accel-Expires`
- `X-Accel-Limit-Rate`

Etag

```
location ~* ^/static/images/$ {
    add_header Cache-Control must-revalidate;
    etag on;
}
```

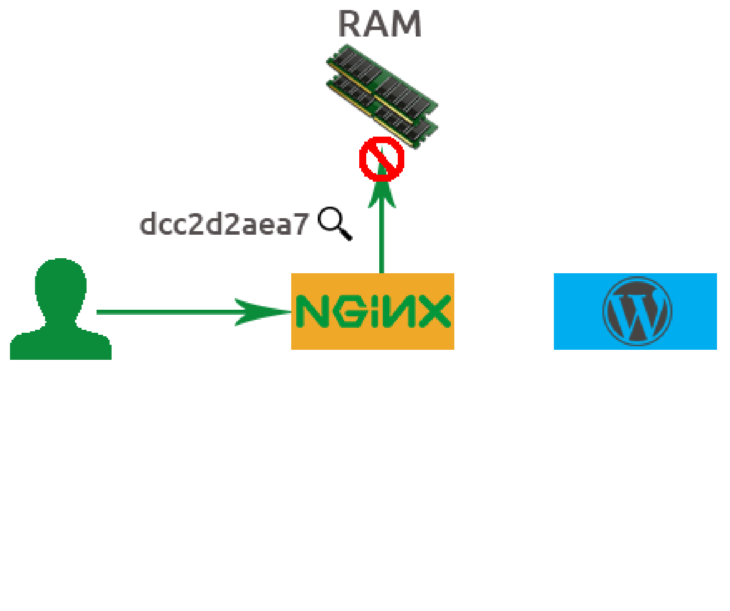
Caching Process Part 1

1. Client makes request
2. Request hits NGINX



Caching Process Part 2

1. NGINX generates hash
2. NGINX checks if hash exists in memory
   - If not, then request proceeds to app

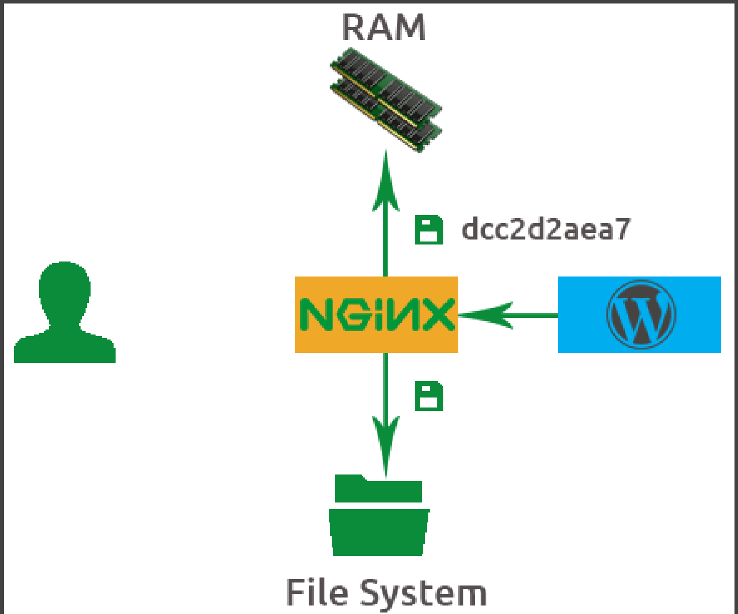


HTTP Header Details

```
#Hash Key
dcc2daea797a0dfd7bac7eec4e33a4a
____________________________________

#md5
(http + example.com + /index.php)

#Request
GET /index.php HTTP.1/1

#Headers
User-Agent: curl/7.35.0
Host: example.com
Accept: * / *

#Body
11101001 10101011 000000000 11101010
```

Caching Process Part 3

1. App sends response
2. Hash saved in memory
3. File saved in file system

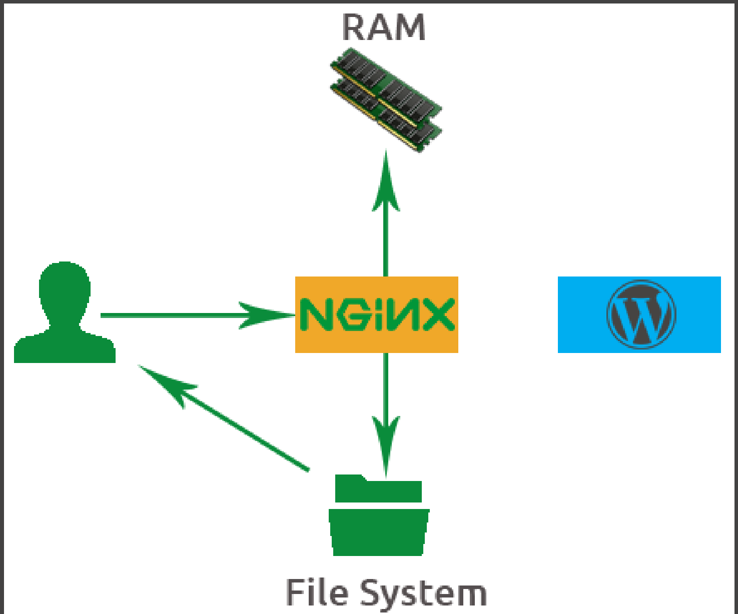


In Memory Details

```
#In Memory
dcc2daea797a0dfd7bac7eec4e33a4a

#In File System
/tmp/cache/a/a4/daea797a0dfd7bac7eec4e33a4a
```

Caching Process Part 4

1. Subsequent request
2. NGINX checks if hash exists in memory
3. Hash points to file
4. Client response sent from cache



NGINX Caching Basics

- proxy_cache_path: sets path
- proxy_cache: invokes the cache
- proxy_cache_key: sets key

```
proxy_cache_path /data/nginx/cache levels=1:2 keys_zone=my_cache:20m inac

server {
    proxy_cache_key $scheme$host$request_uri;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
...
    location /application1 {
        proxy_cache my_cache;
        proxy_cache_valid any 10m;
        proxy_pass https://backend.server;
    }
}
```

Cache Instrumentation

```
add_header X-Cache-Status $upstream_cache_status;
```

`MISS`	Written to cache
`BYPASS`	`proxy_cache_bypass`
`EXPIRED`	Expired entry
`STALE`	Problem with upstream
`UPDATING`	`proxy_use_stale`
`REVALIDATED`	`proxy_cache_revalidate`
`HIT`	Valid, fresh content

Lab 1.1: Reverse Proxy Cache

1. Create a cache directory and a new NGINX conf:

```
$ mkdir -p /data/nginx/cache
$ sudo vim /etc/nginx/conf.d/proxy.conf
```

2. Define a cache path in the `http` context:

```
proxy_cache_path /data/nginx/cache levels=1:2
keys_zone=wordpress_cache:20m inactive=5m;
```

3. In the `server` context, set the cache key and proxy headers. Then instrument the cache:

```
proxy_cache_key $scheme$host$request_uri;
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

add_header X-Cache-Status $upstream_cache_status;
```

Lab 1.2: Reverse Proxy Cache

1. Create a prefix location with an empty string. Then set the `proxy_cache` and the validation for 10 minutes

```
location / {
    proxy_cache wordpress_cache;
    proxy_cache_valid any 10m;
}
```

2. Proxy all port `80` traffic to the WordPress site:

```
proxy_pass http://127.0.0.1:8080/
```

3. Save and reload NGINX. Test the cache using:

```
$ sudo nginx -s reload
$ curl -I http://localhost/
```

NGINX Cache Types

- GET and HEAD with no Set-Cookie response
- Caches based on:
  - raw URL
  - key values

```
#Example
proxy_cache_key $scheme$host$request_uri;
```

- Cache time defined by:
  - X-Accel-Expires
  - Cache-Control
  - Expires

Alternative Caches

- FastCGI
- Memcache
- uwsgi and SCGI

Selective Caching

Separate Cache Placement through keys and regex

```
# Define caches and their locations
proxy_cache_path /mnt/ssd/cache keys_zone=ssd_cache:10m levels=1:2 inact
max_size=700m;
proxy_cache_path /mnt/disk/cache keys_zone=disk_cache:100m levels=1:2 ina
max_size=80G;

# Requests for .mp4 and .avi files go to disk_cache
# All other requests go to ssd_cache
map $request_uri $cache {
    ~\.mp4(\?.*)?$  disk_cache;
    ~\.avi(\?.*)?$  disk_cache;

    default ssd_cache;
}

server {
    # select the cache based on the URI
    proxy_cache $cache;
```

Debugging the Cache

The cache server

```
http {
    ...
    server {
    error_log /path2/to/log debug;
    ...
    }
}
```

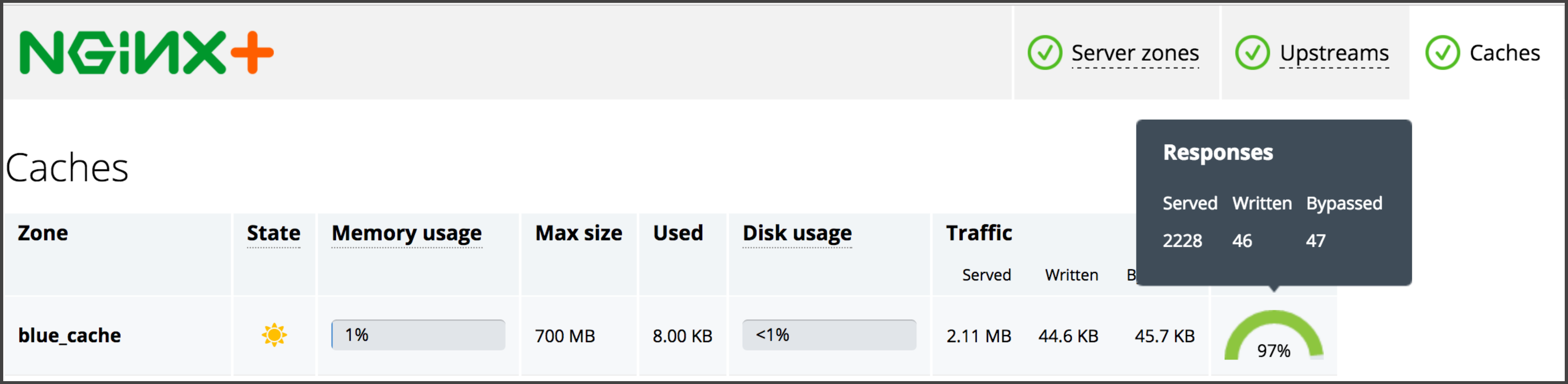
The connection from the load balancer

```
...
events {
    debug_connection 192.168.1.1;
    debug_connection 192.168.10.0/24;
}
```

Documentation: Debugging NGINX

Extended Status

Leverage the status module to view cache stats



Caches

Zone	State	Memory usage	Max size	Used	Disk usage	Traffic			
						Served	Written	B	
blue_cache	☀	1%	700 MB	8.00 KB	<1%	2.11 MB	44.6 KB	45.7 KB	97%

Responses

Served	Written	Bypassed
2228	46	47

Lab 2: Perform Benchmark Test

1. Ensure that `wrk` is installed on the VMs
2. Comment out all `proxy_cache` directives in `cache.conf`
3. Run the following command against your SUT url (ask instructor)

```
wrk -t12 -c400 -d30s <url>
```

Lab 2.5: Cache Wordpress Site

1. Setup extended `status` in order to view cache metrics

```
server {
    listen 9090;
    root /usr/share/nginx/html;

    location = /status {
        status;
    }
}
```

2. Uncomment the `proxy_cache_path` directives
3. Re-run Benchmark Test
4. Log the results as your benchmark

MANAGING CACHED CONTENT

Module Objectives

This module enables you to:

- Identify where cache data is stored
- Modify location (external disc, tmpfs etc.)
- Purge cache entries
- Load instrumented cache data

Persistent Cache

NGINX uses a persistent disk-based cache

Options:

- Load cache data at startup
- Prune the cache over time
- Manually purge content entries

Identifying Location

1. Set content path

```
proxy_cache_path /var/cache/nginx keys_zone=one:10m levels=1:2 max_si
```

2. Define cache key

```
proxy_cache_key $scheme$host$request_uri;
```

3. Get the content into the cache, then check md5

```
$ echo -n "httplocalhost:8080/index.html" | md5sum
6d91blec887b7965d6a926cff19379ba -
```

4. Verify presence of content

```
cat /var/cache/nginx/4/9b/6d91blec887b7965d6a926cff19379ba
```

Lab 3.: Map Files to Disc

1. Map content to disc by running this command

```
$ echo -n "httplocalhost:8080/index.html" | md5sum
```

2. Verify the content is now in the cache directory

```
$ cat /var/cache/nginx/<last 2>/<char of>/<md5 string>
```

3. Restart NGINX

```
$ nginx -s reload
```

Load Previous Instrumentation

```
proxy_cache_path /data/nginx/cache keys_zone=one:10m
loader_files=100;
loader_threshold=200
loader_sleeps=50;
```

- Loads files in blocks of 100
- Takes no longer than 200ms
- Pauses for 50ms, then repeats

Pruning the Cache

The Cache Manager is a background process that operates based on:

- `inactive` parameter
- `max_size` parameter

```
proxy_cache_path /path/to/cache keys_zone=name:size levels=1:2
inactive=time
max_size=size;
```

Lab 4.1: Load Cache Data

1. Stop Nginx by running:

```
$ nginx -s stop
```

2. Use the `loader_files`, `loader_threshold`, and `loader_sleeps` to load previous cache data

```
proxy_cache_path /data/nginx/cache keys_zone=one:10m
    loader_files=100 loader_threshold=200 loader_sleeps=50;
```

3. Restart NGINX

```
$ nginx -s reload
```

4. Check cache capacity in the `status.html` page
5. Run a `curl` request to see if there is a `HIT`

Mounting to tmpfs

1. Create a mount point

```
$ mount -t tmpfs -o size=2G tmpfs /var/cache/nginx
```

2. Match the cache directory in proxy_cache_path or fastcgi_cache_path

```
http {
    proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=one:10m;
    fastcgi_cache_path /var/cache/nginx levels=1:2 keys_zone=one:10
...
}
```

3. Test the RAM directory

```
$ df -ah | grep tmpfs
tmpfs 2.0G 29M 1996M 1% /var/cache/nginx
```

proxy_cache_purge Directive

Allows you to remove full cache entries that match a configured value.

```
server {
    proxy_cache myCache;
    proxy_pass http://localhost:8081;
    proxy_cache_purge $purge_method;
}
```

Purge Methods

Partial Purge

- use `curl` command to send `PURGE HTTP` request, `map` evaluates request and enables the directive

Full Purge

- turn `purger` parameter on in the `proxy_cache_path`, all wildcard pages will also be purged

HTTP PURGE Example

Request:

```
$ curl —X PURGE —D — “http://www.mysite.com"
```

```
# setting the default purge method will only delete matching URLs.
map $request_method $purge_method {
    PURGE 1;
    default 0;
        }
server {
    listen 80;
    server_name www.mysite.com
    proxy_cache myCache;
    proxy_pass http://localhost:8081;
    proxy_cache_purge $purge_method;
}
```

purger Example

Request:

```
$ curl –X PURGE –D – “http://www.mysite.com/*"
```

```
proxy_cache_path /data/nginx/cache levels=1:2 keys=myCache:10m purger=on

server {
    listen 80;
    server_name www.mysite.com;
    location / {

        proxy_cache_purge $purge_method;
    }
}
```

Lab 4.2: Configure Cache Purge

1. Open `myServers.conf` and create the following `map`:

```
map $request_method $purge_method {
        default  0;
        PURGE    1;
}
```

2. Specify the `proxy_cache_purge` directive

```
location / {
        proxy_cache my-cache;
        ....
        proxy_cache_purge $purge_method;
}
```

3. Save and reload NGINX
4. Send the `curl`command using `PURGE`

```
$ curl –X PURGE –I -http://localhost:8080/
```

expires Directive

Expires and Cache-Control response header modification

```
map $sent_http_content_type $expires {

    default                  off;
    application/pdf          42d;
    ~image/                  max;
    css/javascript           modified +24h;
    text/html                epoch;
}

expires $expires;
```

CACHE TUNING

Module Objectives

This module enables you to:

- Interpret and modify headers
- Configure caching resources
- Bypass cache tier

Header Interpretation

Example 1

```
location /images/ {
    proxy_cache my_cache;
    proxy_ignore_headers Cache-Control;
    proxy_cache_valid any 30m;
    ...
}
```

Example 2

```
location /images/ {
    proxy_cache my_cache;
    add_header Cache-Control public;
    ...
}
```

Warning!: add_header Pitfall!

Caching Resources

Directives that control cached responses:

- `proxy_cache_min_uses`
- `proxy_cache_methods`

Caching limit rates:

- `proxy_cache_bypass`
- `proxy_no_cache`

Documentation: Cache Admin Guide

proxy_cache_min_uses

```
server {
    proxy_cache myCache;
    proxy_pass http://localhost:8081;
    proxy_cache_min_uses 5;
}
```

proxy_cache_methods

Syntax:

`proxy_cache_methods $request_method`

```
map $request_method $cache_method {
        default 0;
        GET     1;
        POST    1;
        HEAD    1;
        PUT     0;
}

server {
        proxy_cache_methods $cache_method;
        proxy_cache my_cache;
        proxy_cache valid any 4s;
        proxy_pass http://localhost:8080/;
}
```

proxy_cache_bypass

```
proxy_cache_bypass $cookie_nocache $http_pragma $http_authroization;
```

proxy_cache_no_cache

Syntax:

`proxy_no_cache $arg$arg_comment`

```
map $request_uri $no_cache;
    /default    0;
    /test       1;

server {
    proxy_cache_methods GET HEAD POST;
    proxy_cache my_cache;
    proxy_cache_valid any 10m;
    proxy_no_cache $no_cache;
    proxy_pass http://localhost:8080/;
}
```

proxy_cache_use_stale

```
location / {
    ...
    proxy_cache_use_stale error timeout http_500 http_502 http_503 http_5
}
```

proxy_cache_revalidate

```
location / {
    proxy_cache my_cache;
    proxy_cache_min_uses 3;
    proxy_cache_use_stale error http_500 http_503 http_502;
    proxy_cache_revalidate on;

    proxy_pass http://myUpstream/;
}
```

proxy_cache_lock

```
location / {
    proxy_cache my_cache;
    proxy_cache_min_uses 3;
    proxy_cache_use_stale error http_500 http_503 http_502;
    proxy_cache_revalidate on;

    proxy_cache_lock on;
    proxy_pass http://myUpstream/;
}
```

SwR/SiE

Origin Servers

```
Cache-Control: max-age=3600 stale-while-revalidate=120 stale-if-error=90
```

NGINX Servers

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m
                 max_size=10g inactive=60m use_temp_path=off;

server {
    # ...
    location / {
        proxy_cache my_cache;
        proxy_cache_use_stale updating;
        proxy_cache_background_update on;
        proxy_pass http://my_upstream;
    }
}
```

Cache-Control Review

- `proxy_cache_valid`
- `proxy_cache_bypass`
- `proxy_no_cache`
- `Cache-Control public`

Lab 5.1: Create Cache Params

1. Create a config in `/etc/nginx/conf.d/` called `cache.params.conf`
2. Create a `map` that ONLY caches `GET`, `HEAD`, and `POST`, all other `$request_methods` set to 0
3. Create another `map` for `expires` duration for each `$content-type`:
    - application/pdf
    - images
    - css/javascript
    - text/html

Lab 5.2: Control Responses

1. In `cache.params.conf`, create a `map` that will NOT cache the following responses: 404, 502-504.
2. Set `default` to 0
3. Then create a `map` that will indicate when the cache should use "stale" content based on `proxy_next_upstream` responses. For example:

```
map $proxy_next_upstream $stale_methods {
    error           1;
    timeout         1;
    invalid_header  1;
    ...
    default         0;
}
```

Lab 5.3: Set Cache Params

1. Open `cache.conf` and edit the `server` context:

```
server {
    listen 80;
    include /etc/nginx/conf.d/cache.params.conf;
    ...
    proxy_no_cache $no_cache;
    proxy_cache_methods $cache_methods;
    expires $expires;
    proxy_cache_use_stale $stale_methods;
}
```

2. Save and Reload NGINX
3. Run the following tests:

```
curl -I http://localhost/test
curl -I http://localhost/index.html
curl -I http://localhost/mycontent.html
```

CACHE SCALING

Module Objectives

This module enables you to:

- Enable microcaching
- Create and Deploy Cache Placement Strategies
- Optimize read and write operations
- Create high-capacity/highly-available caches

Microcaching

Benefits:

- Improves web performance
- Reduces load on origin servers

Drawbacks:

- Depends on cacheability of content
- Spike on origin server after entry expires

Microcaching Scenarios

- Front page of busy blog or news site
- RSS feed of recent information
- Status page of a CI build platform
- Calendar data
- Personalized dynamic content on client side

Microcaching Example

```
proxy_cache_path /tmp/cache keys_zone=cache:10m levels=1:2 inactive=600s
server {
    listen external-ip:80;  # External IP address
    proxy_cache cache;
    proxy_cace_valid 200 1s;
    status_zone wordpress; # NGINX Plus status monitoring

    location / {
        proxy_http_version 1.1; # Always upgrade to HTTP/1.1
        proxy_set_header Connection ""; # Enable keepalives
        proxy_set_header Accept-Encoding ""; # Optimize encoding
        proxy_pass http://wordpress-upstreams;
    }
}

upstream wordpress-upstreams {
    zone wordpress 128k;
    keepalive 20; # Keepalive pool to upstream
```

Cache Placement Strategies

- Single Disk
- Mirror
- Stripe
- Hash

Single Disk

```
proxy_cache_path /tmp/cache keys_zone=cache:10m levels=1:2 inactive=600s

                server {
                ...
                proxy_cache cache
                proxy_cache valid 200 15s;
        }
```

Mirror

```
$ lvcreate -L 50G -m1 -n mirrorlv vg0
```

Documentation: Create Mirrored Volume

Stripe

```
$ lvcreate -i3 -I4 -L1G -nmy_logical_volume my_volume_group
lvcreate -- rounding 1048576 KB to stripe boundary size 1056768 KB / 258
```

Documentation: Create Striped Logical Volume

Hash

```
proxy_cache_path /mnt/disk1/cache keys_zone=disk1:100m levels=1:2 inacti
                 max_size=5G use_temp_path=off;
proxy_cache_path /mnt/disk2/cache keys_zone=disk2:100m levels=1:2 inacti
                 max_size=5G use_temp_path=off;

split_clients $request_uri $cache {
    50%     disk1;
    *       disk2;
}

server {
    listen localhost:80;
    proxy_cache $cache;
    status_zone loadbalancer;
...
}
```

Testing Placement Strategies

- Use `iostat` to monitor I/O on individual disks
- Test both cloud and bare-metal servers
- Make sure caches are flushed and empty before each test run

Documentation: Cache Placement Strategy

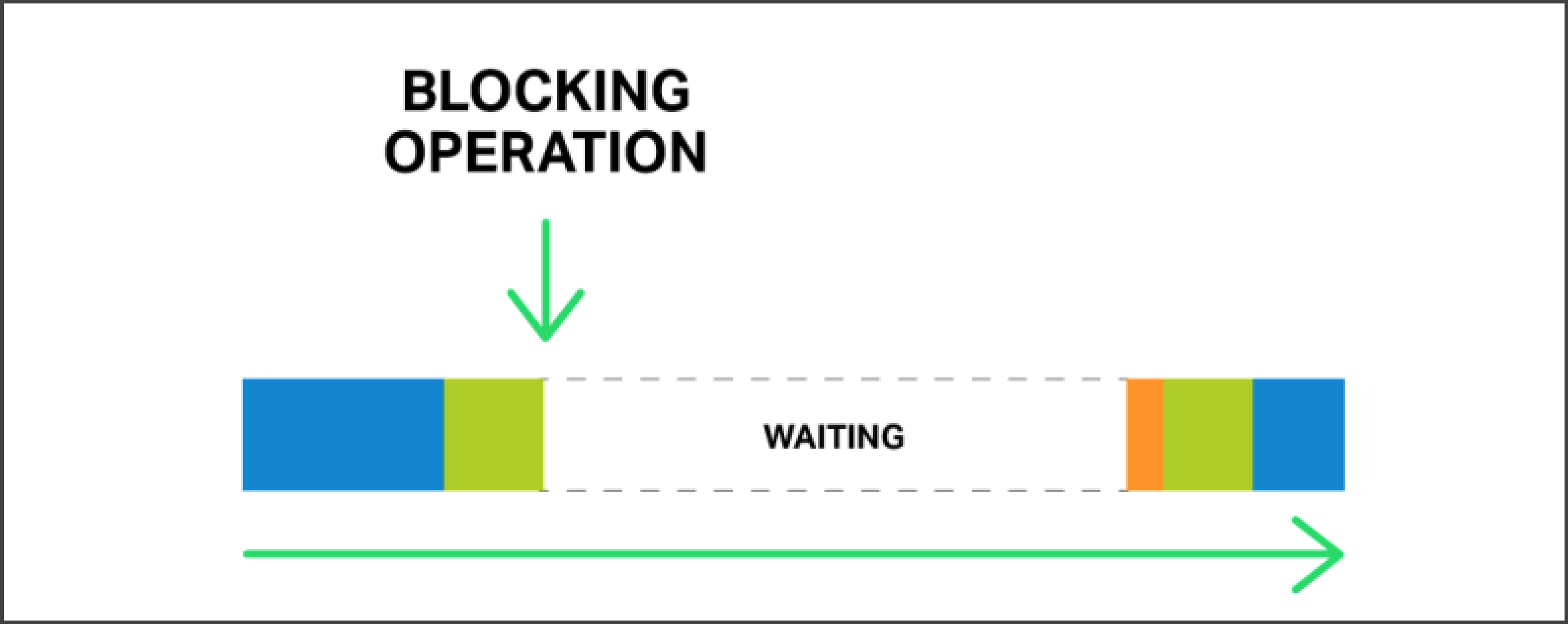
Optimizing Read Operations

- `aio`
- `directio`
- `thread_pools`

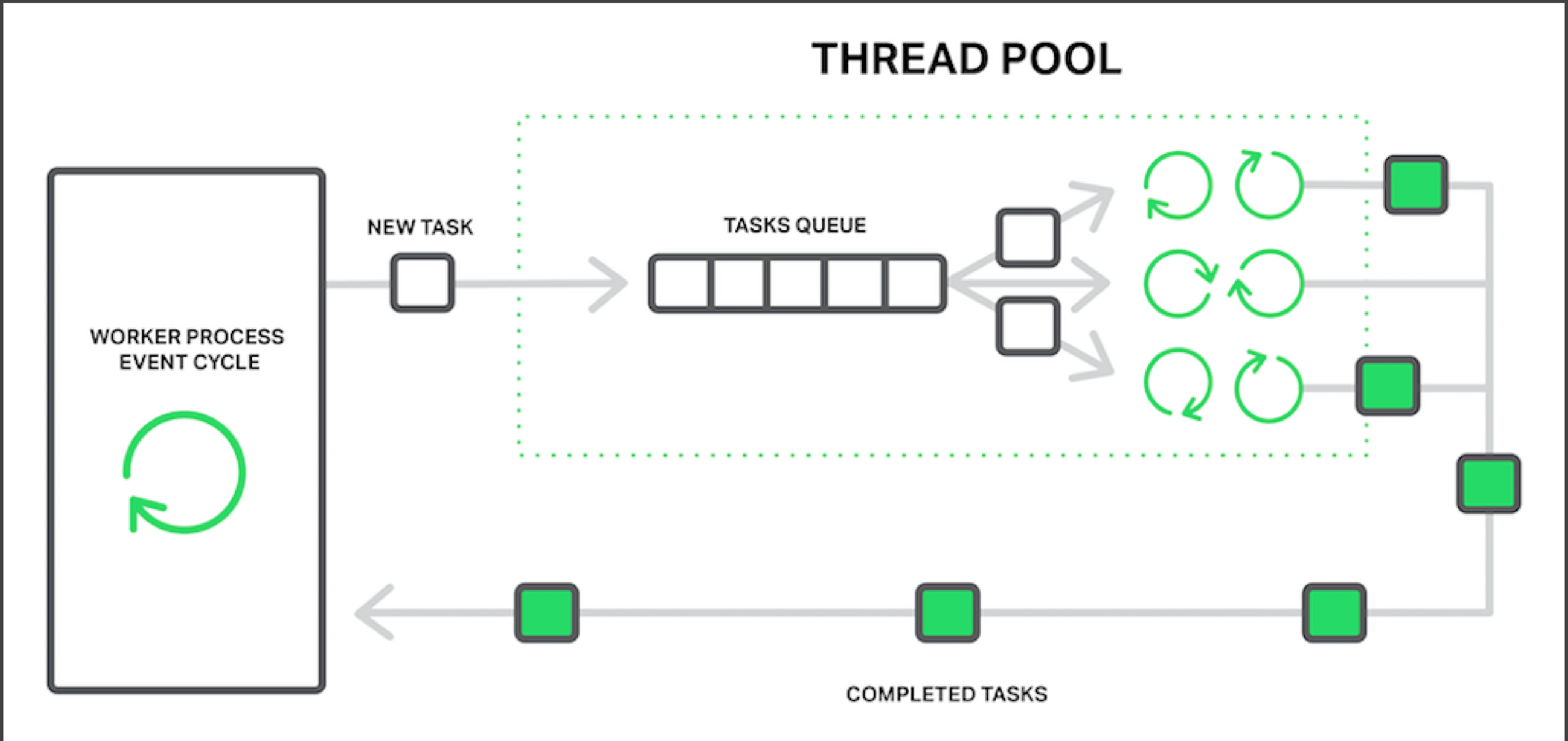
Async I/O

```
thread_pool io_pool threads=16;
http{
…......
   location /data{
     sendfile   on;
     aio       threads=io_pool;
   }
}
```

Blocking Operations



thread_pools



Byte Range Requests

Problem: Subsequent requests spawn new cache-fill operations during long cache-fill.

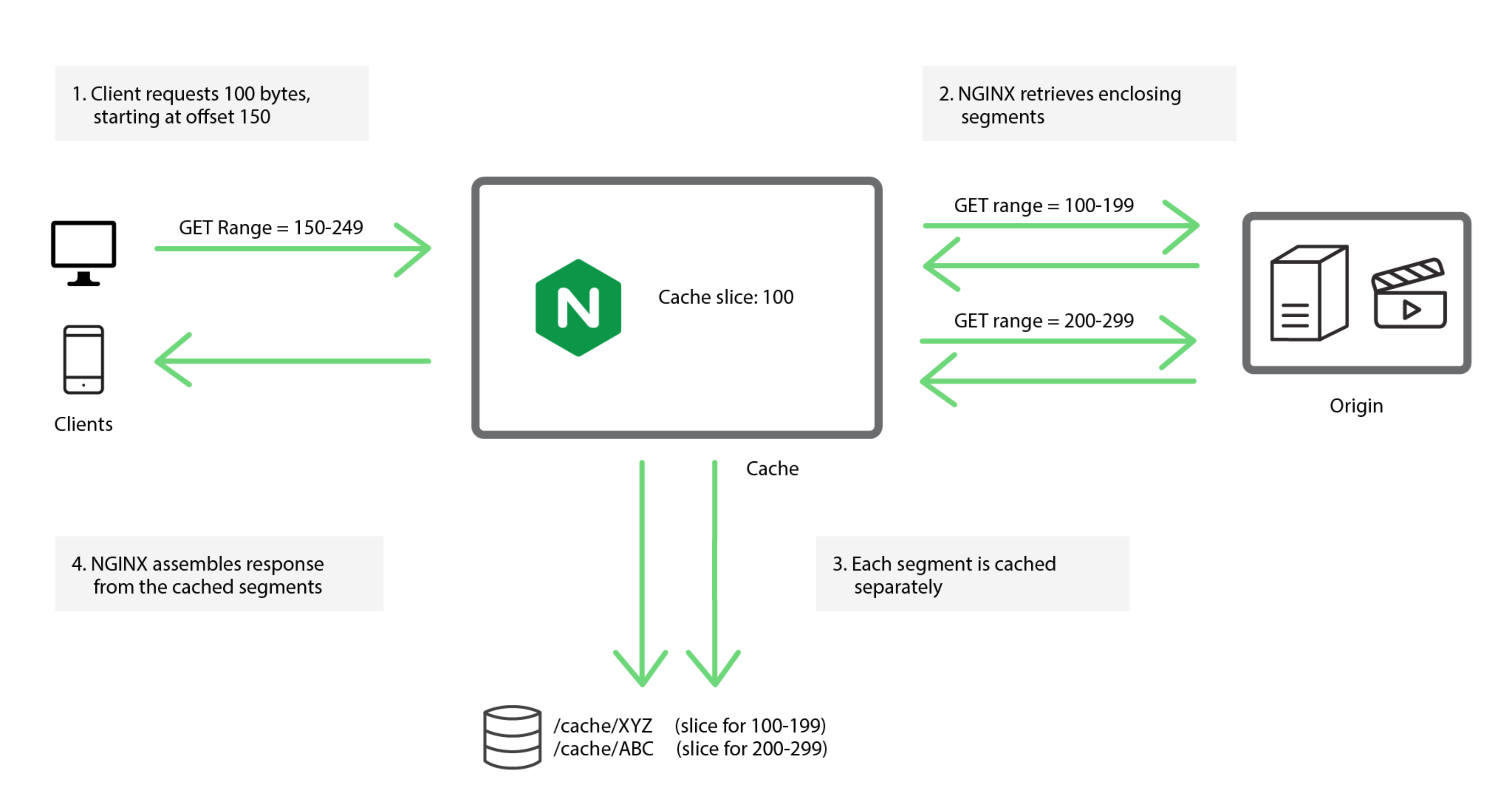
Solution: Cache lock or slicing

Lock a Single Fill

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;

server {
    listen 80;

    proxy_cache mycache;
    proxy_cache_valid 200 600s;
    proxy_cache_lock on;

    # Immediately forward requests to the origin if we are filling the ca
    proxy_cache_lock_timeout 0s;

    # Set the 'age' to a value larger than the expected fill time
    proxy_cache_lock_age 200s;

    proxy_cache_use_stale updating;

    location / {
```

Slice-by-Slice

Use the Cache Slice module to optimize bandwidth during long cache-fill operations



1. Client requests 100 bytes, starting at offset 150

2. NGINX retrieves enclosing segments

3. Each segment is cached separately

4. NGINX assembles response from the cached segments

slice Example

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;

server {
    listen 80;

    proxy_cache mycache;

    slice              1m;
    proxy_cache_key    $host$uri$is_args$args$slice_range;
    proxy_set_header   Range $slice_range;
    proxy_http_version 1.1;
    proxy_cache_valid  200 206 1h;

    location / {
        proxy_pass http://origin:80;
    }
}
```

Splitting Across Disks

```
proxy_cache_path /path/to/hdd1 levels=1:2 keys_zone=my_cache_hdd1:10m
                 max_size=10g inactive=60m use_temp_path=off;
proxy_cache_path /path/to/hdd2 levels=1:2 keys_zone=my_cache_hdd2:10m
                 max_size=10g inactive=60m use_temp_path=off;

split_clients $request_uri $my_cache {
              50%          “my_cache_hdd1”;
              50%          “my_cache_hdd2”;
}

server {
    ...
    location / {
        proxy_cache $my_cache;
        proxy_pass http://my_upstream;
    }
}
```

Lab 6: Split Across Directories

1. Create two cache directories to emulate hard disks

```
$ sudo mkdir -p /tmp/cache1 /tmp/cache2
```

2. Use `split_clients` to spread cache writes

```
split_clients $request_uri $cache {
                50%          cache1;
                *            cache2;
}
```

3. Run the followind dynamic `curl` request

```
$ for i in `seq 1 100` ; do curl -s -o /dev/null
-w "%{http_code}" http://<external_ip>/\?$i ; done
```

4. Run `top` and press 'c' to see paths

```
$ top -u nginx
$ c
```

Cache Clusters

High Cacpacity

- Sharded Cache
- "Hot" level Cache

High Availability:

- Shared Cache
  - keepalived (VRRP)
  - All-Active GCE

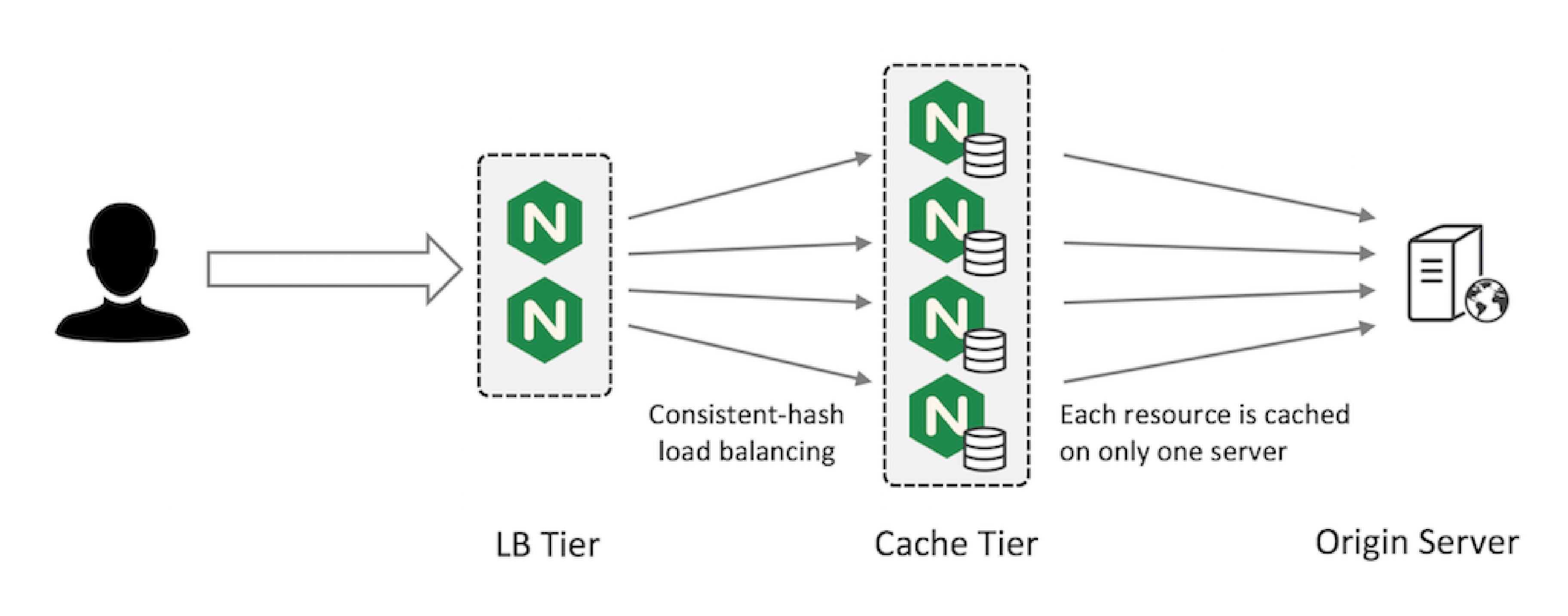
No Shared Disk?

NGINX Plus is sensitive to disk latency, potentially overwhelmed by thread volumes, and requires cluster-wide locks that could result in overlapping cache operations

Sharded Cache

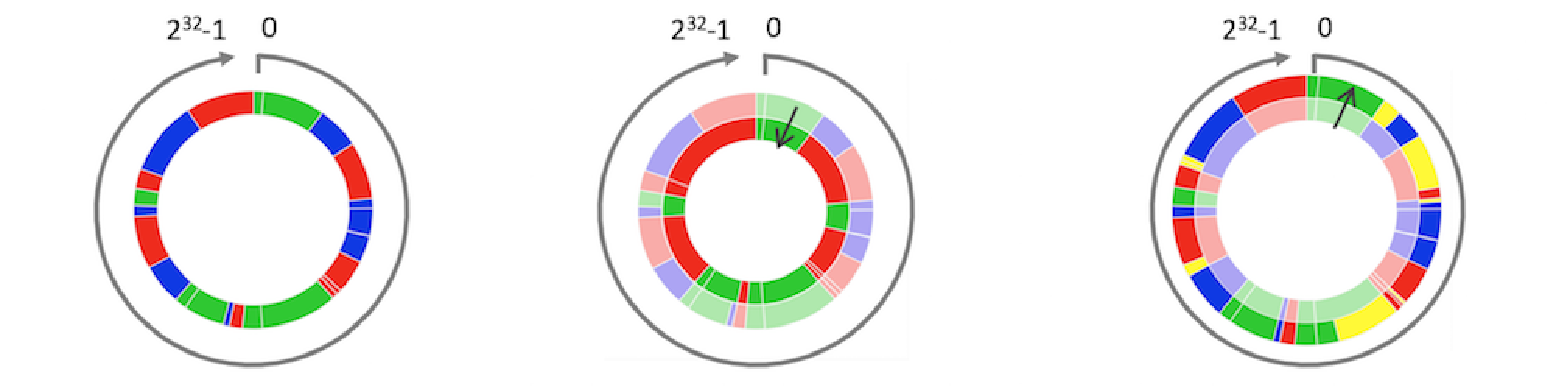
Primary Use Cases:

- High Capacity—partitioned across multiple servers
- High Performance—minimizes origin server load


Consistent-hash load balancing
Each resource is cached on only one server
LB Tier
Cache Tier
Origin Server

Fault Tolerance Scenarios

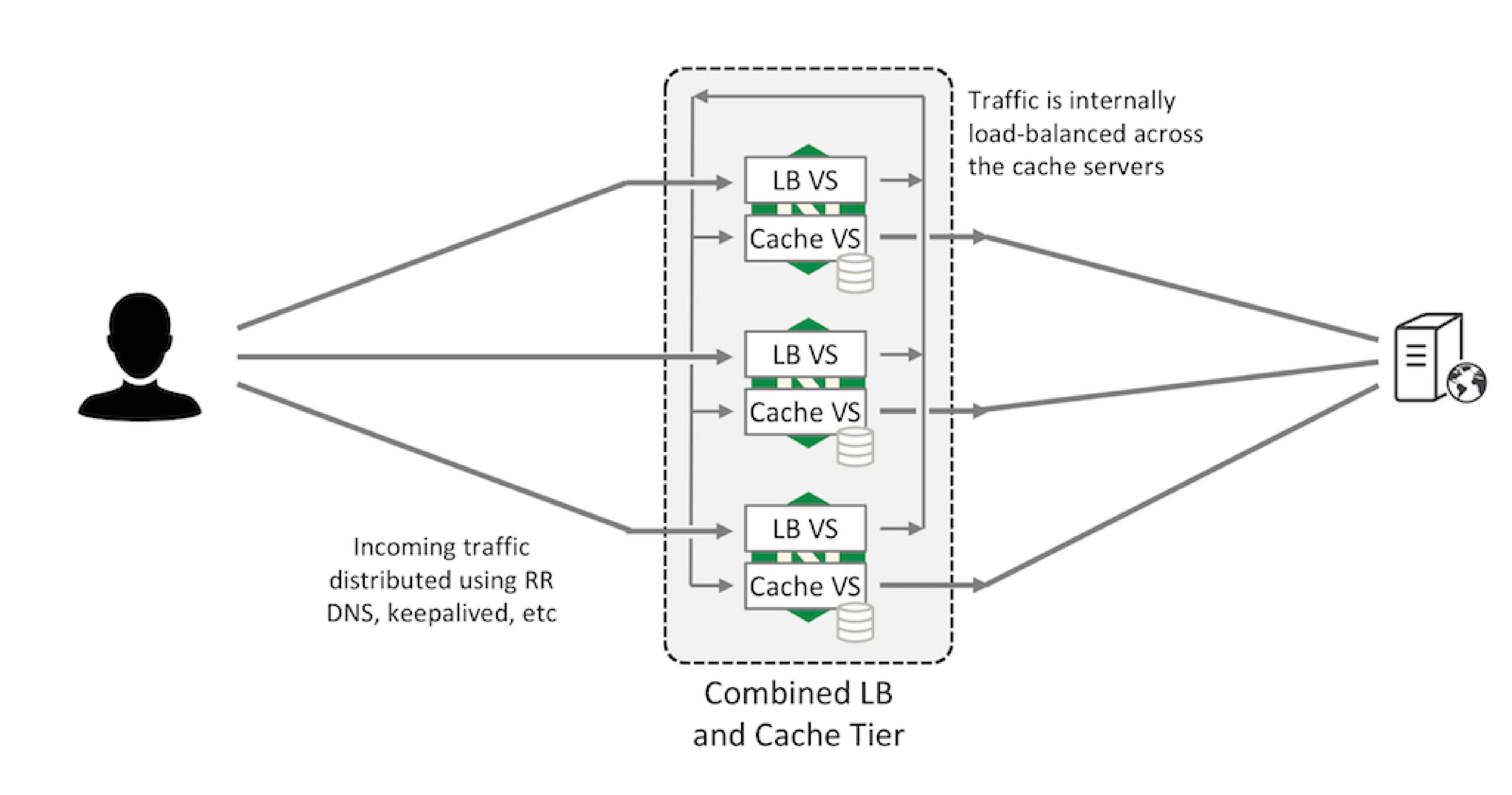
One node fails, only 1/N cached data is 'lost'. New nodes automatically partition entries



Consistent Hashing

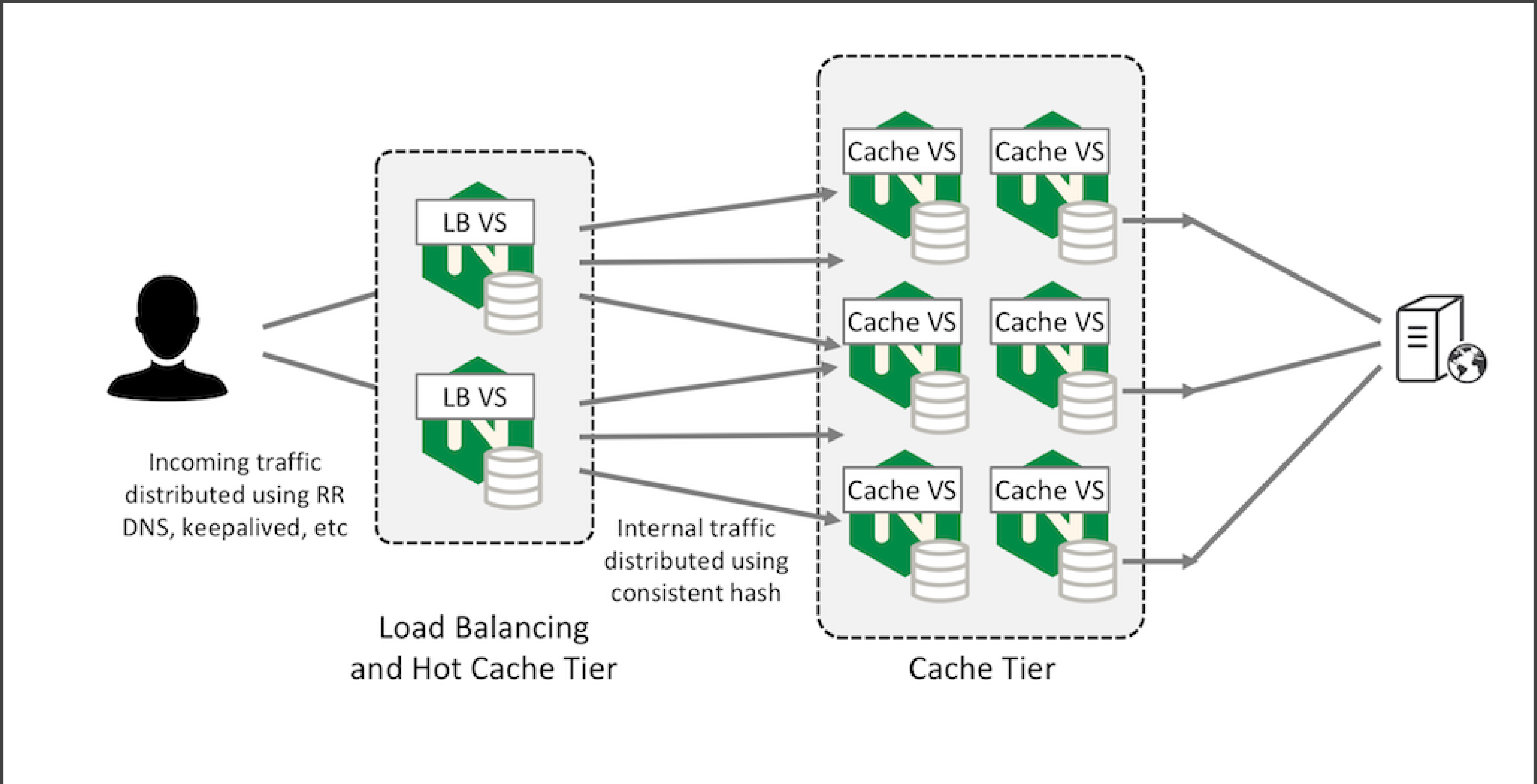
```
upstream cache-servers {
        hash $scheme$proxy_host$request_uri consistent;

        server cache-server1;
        server cache-server2;
        server cache-server3;
}
```

Combining LB and Cache Tiers



Traffic is internally load-balanced across the cache servers
LB VS
Cache VS
LB VS
Cache VS
LB VS
Cache VS
Incoming traffic distributed using RR DNS, keepalived, etc
Combined LB and Cache Tier

"Hot" Cache


Incoming traffic distributed using RR DNS, keepalived, etc
LB VS
LB VS
Load Balancing and Hot Cache Tier
Internal traffic distributed using consistent hash
Cache VS
Cache VS
Cache VS
Cache VS
Cache VS
Cache VS
Cache Tier

Demo 7.1: Setting up the Cache Tier

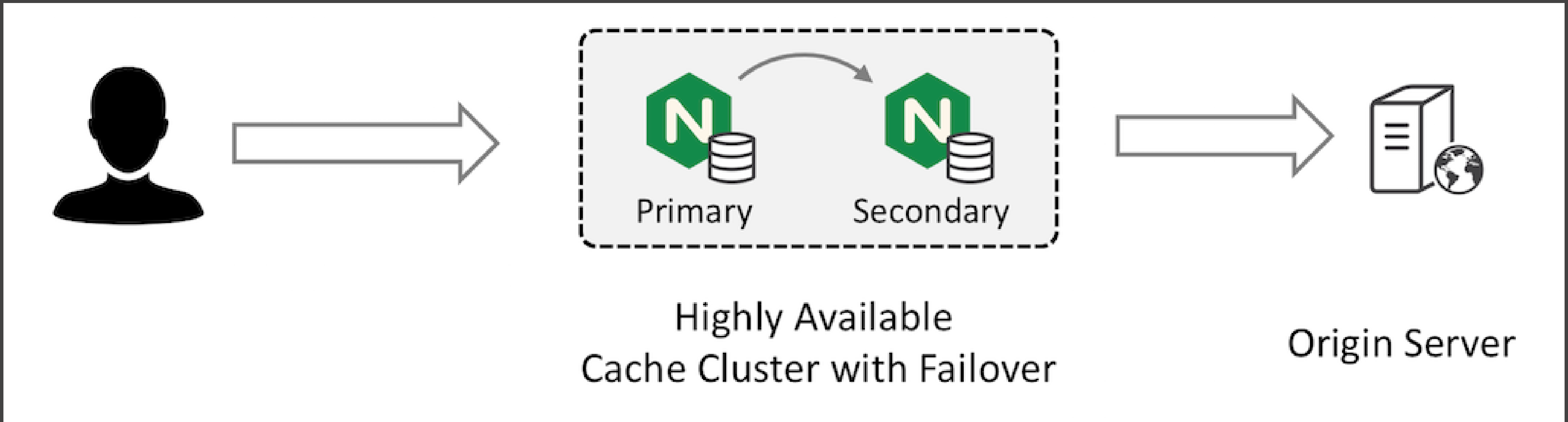
Prerequisites

- Load Balancer with NGINX Plus
- Application Server (origin)
- At least two NGINX Plus Cache Servers
  - Extended Status enabled to see the cache fill

Demo 7.2: Sharding the Cache

1. Enable `consistent hash` on the cache upstream
2. Make dynamic requests to spread across nodes
3. Stop NGINX processes on one of the cache servers, note that server responses are still sent from the second cache

High Availability Cache



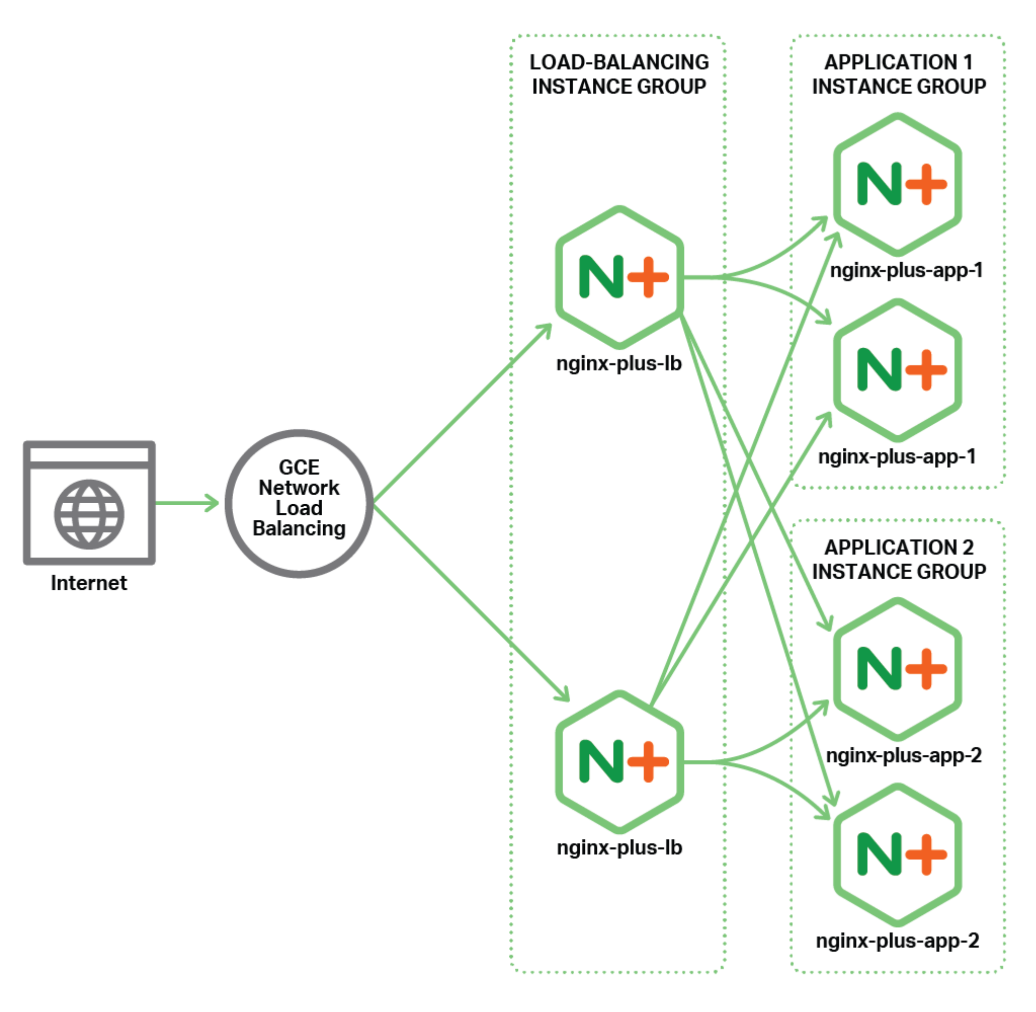
Primary Cache

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
    status_zone mycache;          # for NGINX Plus status dashboard
    listen 80;
    proxy_cache mycache;
    proxy_cache_valid 200 15s;

    location / {
        proxy_pass http://secondary;
    }
}

upstream secondary {
    zone secondary 128k;          # for NGINX Plus status dashboard
    server 192.168.56.11;         # secondary
    server 192.168.56.12 backup;  # origin
}
```

Secondary Cache

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
    status_zone mycache;          # for NGINX Plus status dashboard
    listen 80;
    proxy_cache mycache;
    proxy_cache_valid 200 15s;

    location / {
        proxy_pass http://origin;
    }
}

upstream origin {
    zone origin 128k;            # for NGINX Plus status dashboard
    server 192.168.56.12;        # origin
}
```

GCE HA Solution

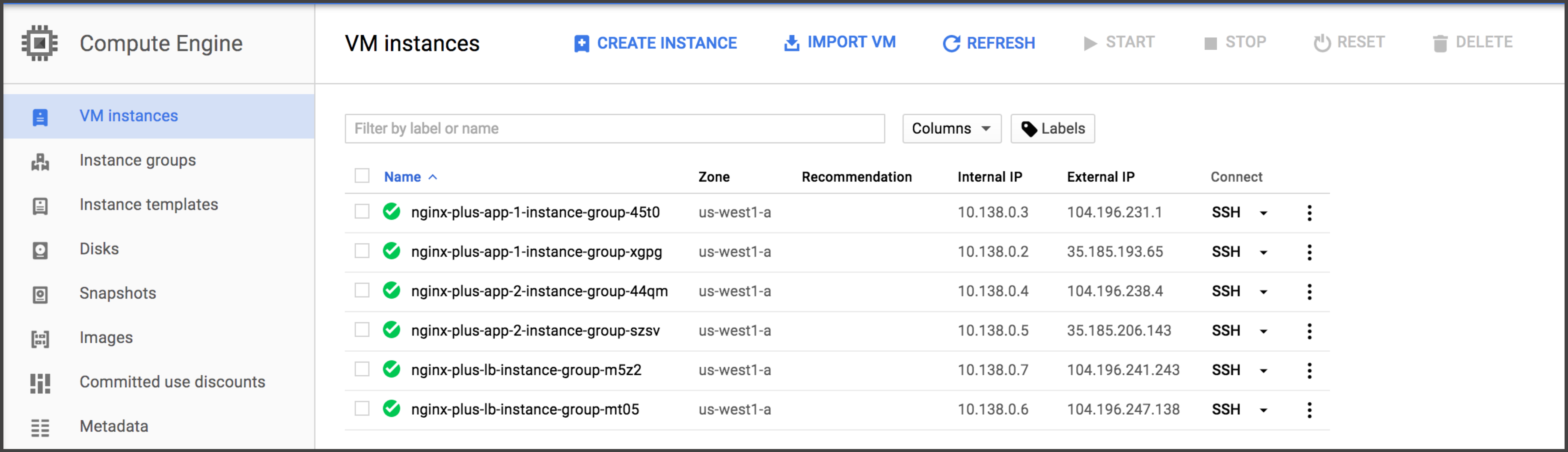

LOAD-BALANCING INSTANCE GROUP
APPLICATION 1 INSTANCE GROUP
APPLICATION 2 INSTANCE GROUP
Internet
GCE Network Load Balancing
N+
nginx-plus-lb
nginx-plus-lb
nginx-plus-app-1
nginx-plus-app-1
nginx-plus-app-2
nginx-plus-app-2

Advantages:

1. Detects changes
2. Active checking
3. Automatic recovery

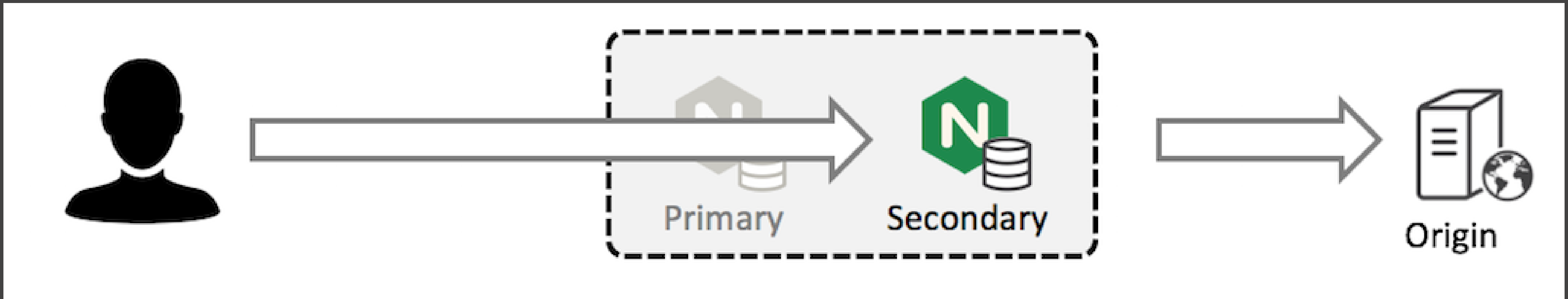
Deployment Guide: All-Active GCE Load Balancer

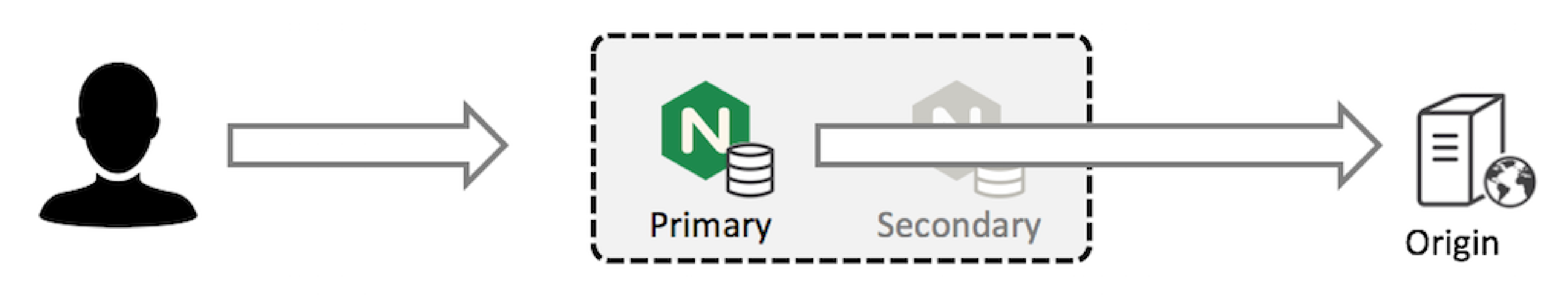
Demo 8.1: HA Shared Cache



□	Name	Zone	Recommendation	Internal IP	External IP	Connect	
□	✓ nginx-plus-app-1-instance-group-45t0	us-west1-a		10.138.0.3	104.196.231.1	SSH ▾	⋮
□	✓ nginx-plus-app-1-instance-group-xgpg	us-west1-a		10.138.0.2	35.185.193.65	SSH ▾	⋮
□	✓ nginx-plus-app-2-instance-group-44qm	us-west1-a		10.138.0.4	104.196.238.4	SSH ▾	⋮
□	✓ nginx-plus-app-2-instance-group-szsv	us-west1-a		10.138.0.5	35.185.206.143	SSH ▾	⋮
□	✓ nginx-plus-lb-instance-group-m5z2	us-west1-a		10.138.0.7	104.196.241.243	SSH ▾	⋮
□	✓ nginx-plus-lb-instance-group-mt05	us-west1-a		10.138.0.6	104.196.247.138	SSH ▾	⋮

Failover Scenenarios





Demo 8.2: Testing Failover

1. Configure origin server

```
access_log /var/log/nginx/access.log;
location / {
    return 200 "It's now $time_local\n";
}
```

2. Configure cache validation

```
proxy_cache_valid 200 15s;
```

3. Verify cache behavor

```
$ while sleep 1 ; do curl http://<external frontend lb ip>/
```

4. Verfiy failover behavior (2nd scenario)

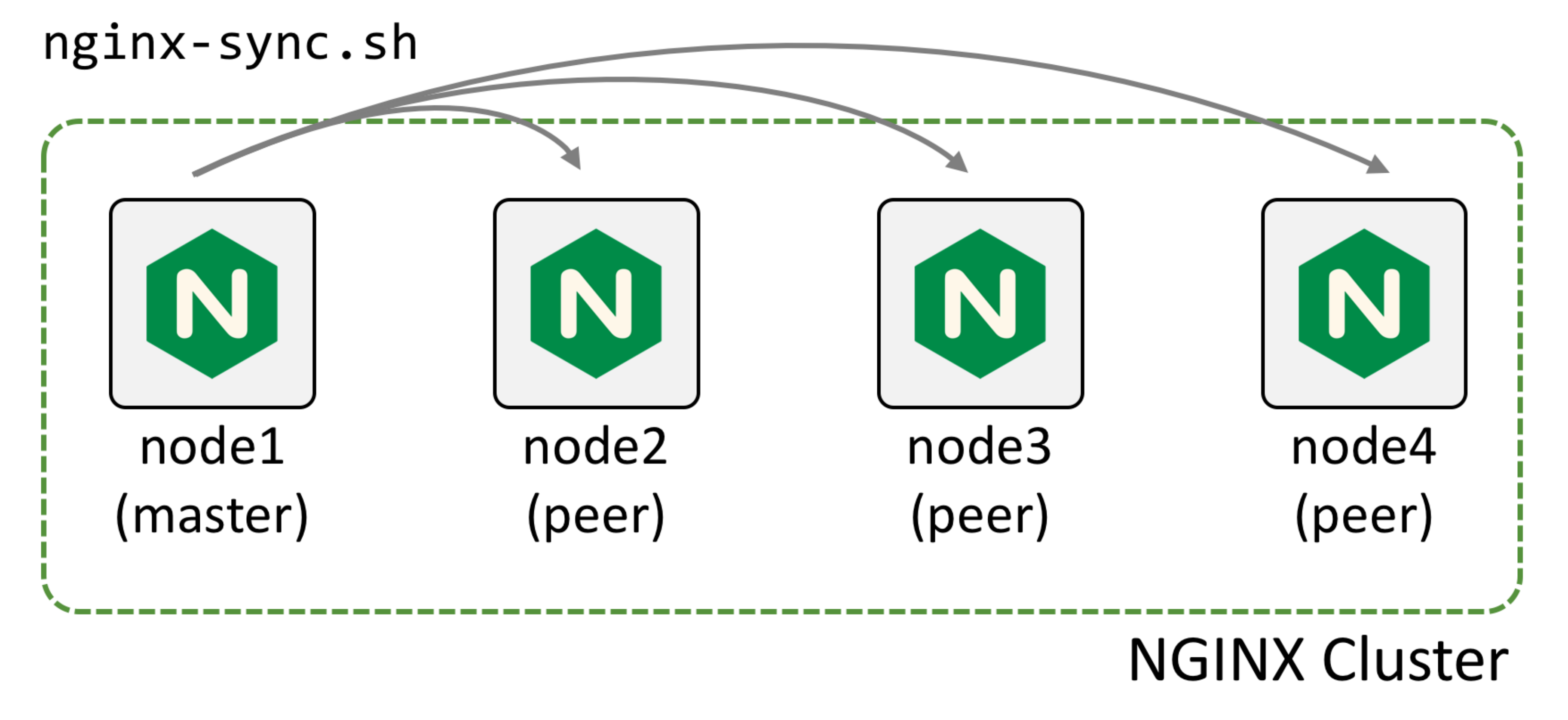
```
$ nginx -s stop
# Inspect log on origin server
tail -f /var/log/nginx/access.log
```

Timing Cache Updates

`expires` headers that are shared by mutliple instances, could result in time stamp mismatch. It's recommended to shorten the cache timeout on the secondary server

nginx-sync

Share configurations in an HA cluster



Documentation: nginx-sync

ADDITIONAL RESOURCES

Further Information

- NGINX Documentation
- NGINX Admin Guides
- NGINX Blog

Q&A

- Survey!
- Sales: nginx-inquiries@nginx.com