VDT 2024
BÁO CÁO GIỮA KỲ

Lê Minh Việt Anh

Đại học Bách khoa Hà Nội

MỤC LỤC

Tóm tắt nội dung

Nội dung báo cáo gồm hai phần chính:

Phần 1: Thực hành xây dựng 3 tier web app đơn giản và triển khai hệ thống bằng các công cụ DevOps

Các kết quả đạt được trong bài thực hành sẽ được trình bày, các file và ảnh yêu cầu được đẩy lên repo báo cáo.

Phần 2: Nghiên cứu về cân bằng tải trong điện toán đám mây

Trình bày bối cảnh, định nghĩa về cân bằng tải, tìm hiểu về cân bằng tải trên đám mây và tìm hiểu sâu về các thuật toán cân bằng tải, …

Phần 1: Thực hành

Phát triển một 3-tier web đơn giản

Công nghệ sử dụng:

Web: react js

Api: node js

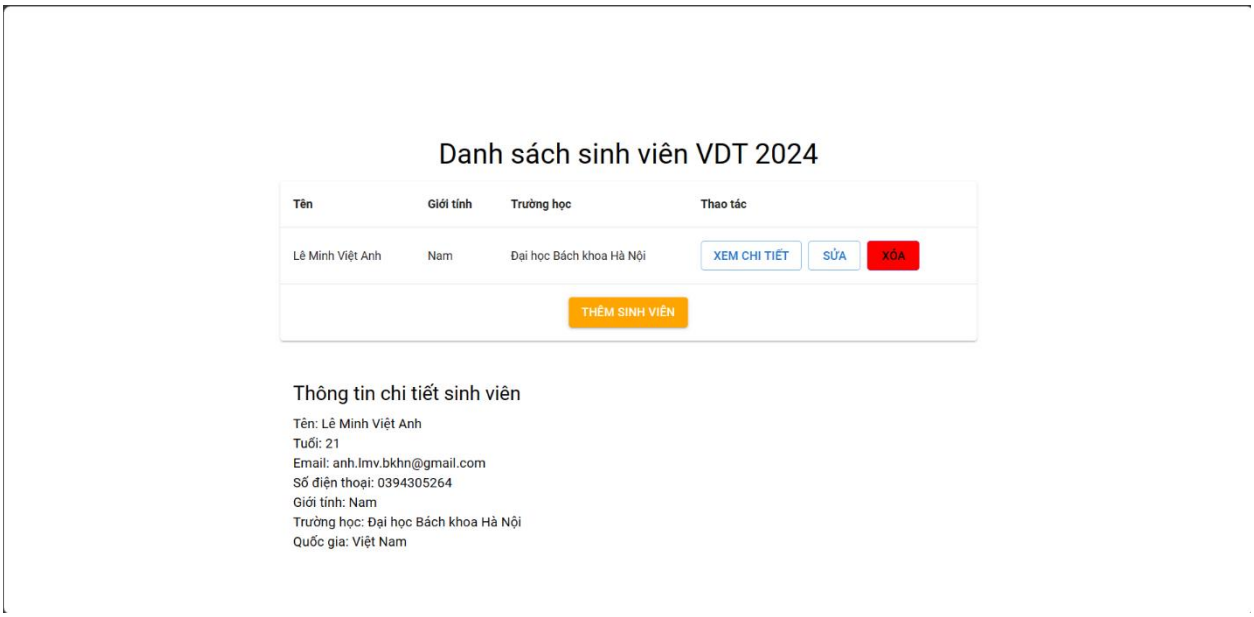
Test: jest, super test

Db: MongoDB

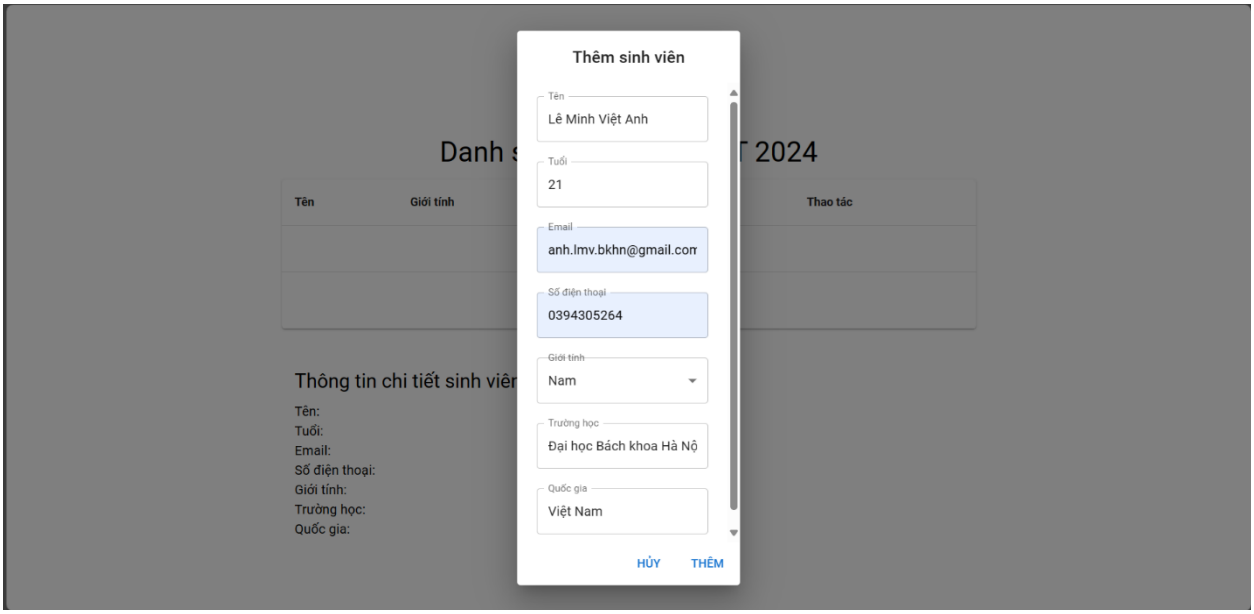
Maintain trên hai repo:

Web: https://github.com/shukuchicoding/vdt2024_web.git

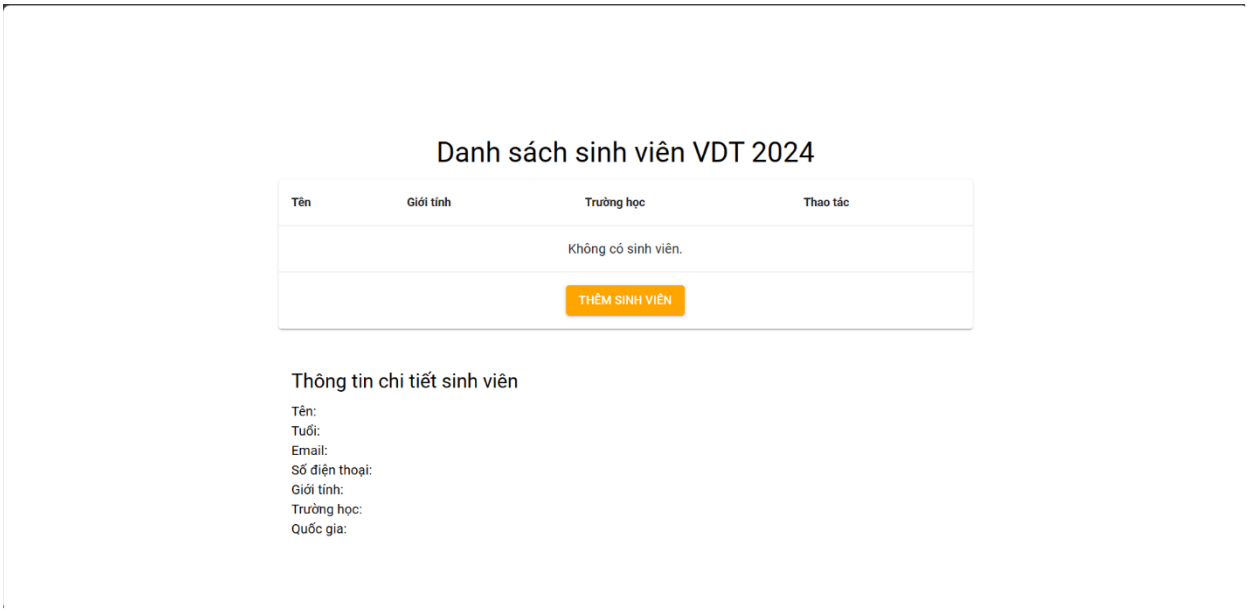
Api: https://github.com/shukuchicoding/vdt2024_api.git



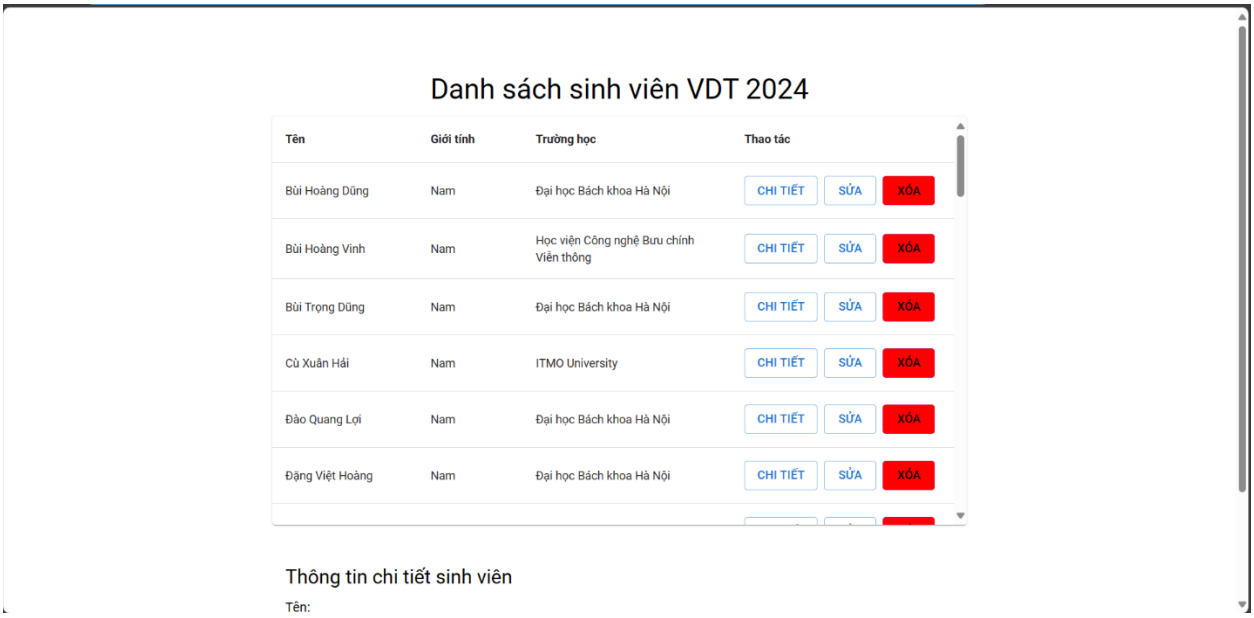
Ảnh 1: Giao diện chính



Ảnh 2: Giao diện add



Ảnh 3: Giao diện ban đầu



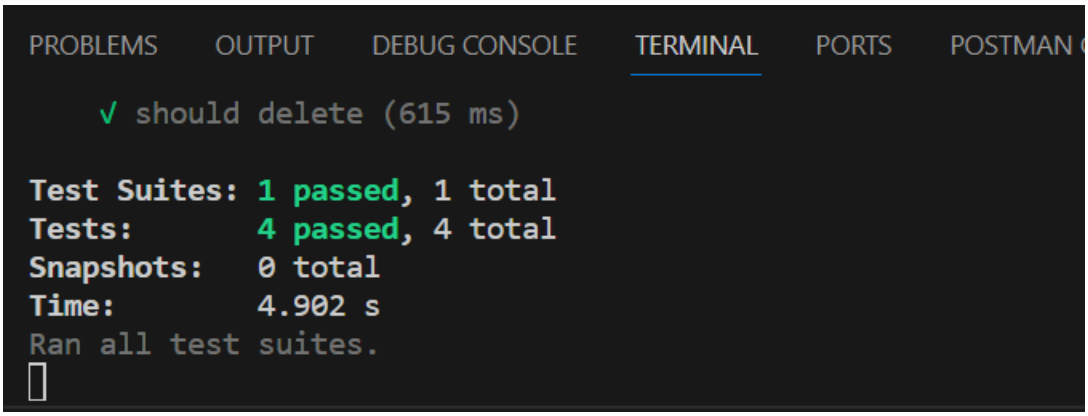
Ảnh 4: Giao diện sau khi thêm đầy đủ các thành viên

Thực hiện chức năng chỉnh sửa thông tin sinh viên:

Đổi trường học của sinh viên Lê Minh Việt Anh từ Đại học Bách khoa Hà Nội thành Đại học Nông nghiệp.

Ảnh 5: Hiển thị thông tin chi tiết, sau khi update

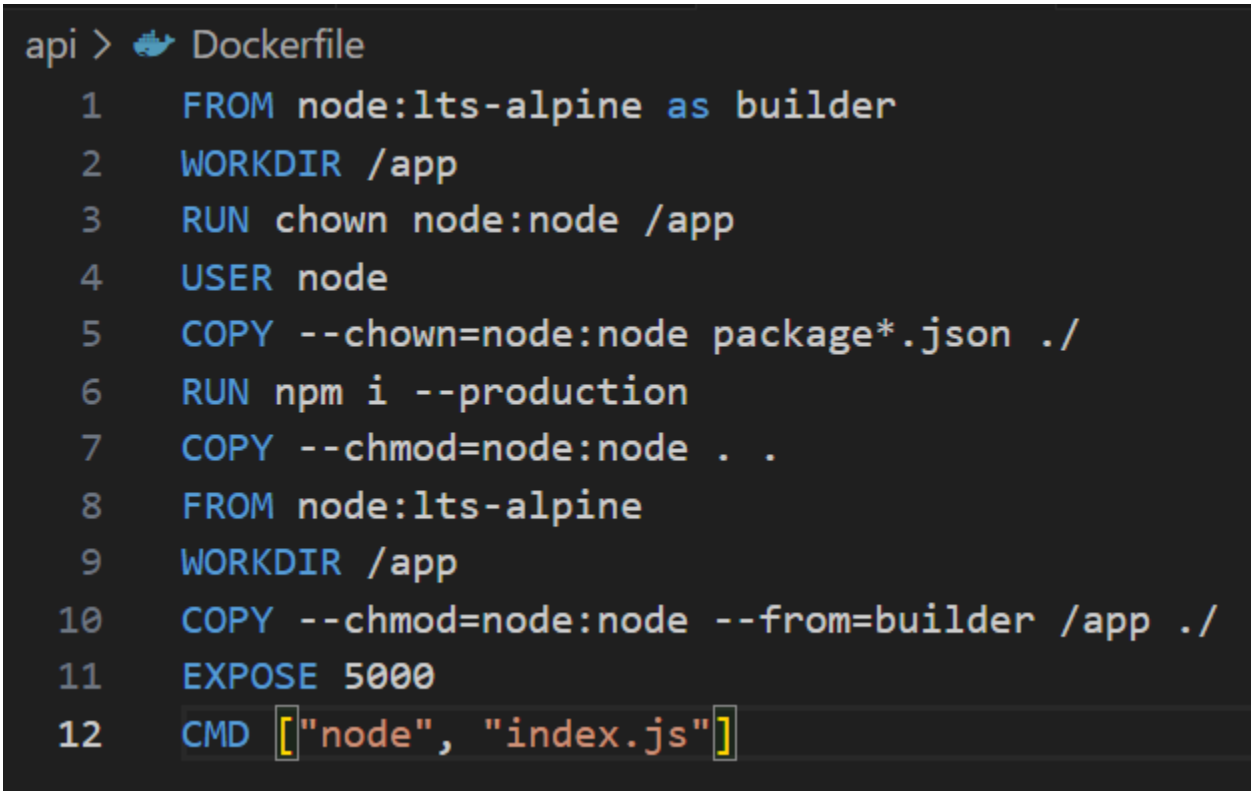
Xây dựng 4 test case đơn giản cho các chức năng: GET, UPDATE, POST, DELETE



Ảnh 6: Passed test cases

Triển khai web application sử dụng devops tools & practices

Containerization



Ảnh 7: Dockerfile cho API

Các kỹ thuật áp dụng: Multi stage, chỉ định quyền user node để tăng tính an toàn cho hệ thống, tối ưu câu lệnh RUN (không cài các công cụ dev trong package json)

web > Dockerfile

```dockerfile
#1. Build
FROM node:lts-alpine as builder
WORKDIR /app
RUN chown node:node /app
USER node
COPY --chown=node:node package*.json ./
RUN npm install
COPY --chmod=node:node . .
RUN npm run build
CMD ["node"]
#2. Serve - production
FROM nginx:stable-alpine
COPY --chmod=node:node --from=builder /app/build /usr/share/nginx/html
COPY --chmod=node:node --from=builder /app/nginx/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 3000
CMD ["nginx", "-g", "daemon off;"]
```

Ảnh 8: Dockerfile cho WEB

Các kỹ thuật áp dụng: Multi-stage build (build stage và production stage), optimized RUN instruction, layer caching

```
View build details: docker-desktop:///dashboard/build/default/default/ge0bceg7j8j2b307tud5167dt
PS D:\VDT_2024\Midterm\api> docker build -t vdt-api:1.0.0 .
[+] Building 0.0s (0/0)  docker:default
2024/05/22 23:53:28 http2: server: error reading preface from client //./pipe/docker_engine: file has already been close
[+] Building 14.3s (12/12) FINISHED                                                                    docker:default
 => [internal] load build definition from Dockerfile                                                          0.0s
 => => transferring dockerfile: 269B                                                                          0.0s
 => [internal] load metadata for docker.io/library/node:lts-alpine                                            2.3s
 => [auth] library/node:pull token for registry-1.docker.io                                                   0.0s
 => [internal] load .dockerignore                                                                             0.0s
 => => transferring context: 75B                                                                              0.0s
 => [internal] load build context                                                                             1.1s
 => => transferring context: 476.39kB                                                                         1.0s
 => [1/6] FROM docker.io/library/node:lts-alpine@sha256:291e84d956f1aff38454bbd3da38941461ad569a185c20aa289f71f37  0.0s
 => CACHED [2/6] WORKDIR /app                                                                                 0.0s
 => CACHED [3/6] RUN chown node:node /app                                                                     0.0s
 => CACHED [4/6] COPY --chown=node:node package*.json ./                                                      0.0s
 => CACHED [5/6] RUN npm i --production                                                                       0.0s
 => [6/6] COPY --chmod=node:node . .                                                                         10.0s
 => exporting to image                                                                                        0.6s
 => => exporting layers                                                                                       0.6s
 => => writing image sha256:da2963e9793a586534216e41dc8ef9281d52e2a479a851d2b8172bef39530290                  0.0s
 => => naming to docker.io/library/vdt-api:1.0.0                                                              0.0s

View build details: docker-desktop://dashboard/build/default/default/xzyb69xoomq3aj31mtrm2xp8v

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS D:\VDT_2024\Midterm\api>
```

Ảnh 9: Output câu lệnh build của API Dockerfile

```
View build details: docker-desktop://dashboard/build/default/default/xzyb69xoomq3aj31mtrm2xp8v

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS D:\VDT_2024\Midterm\api> docker history vdt-api:1.0.0
IMAGE          CREATED             CREATED BY                                      SIZE      COMMENT
da2963e9793a   29 seconds ago      CMD ["node" "index.js"]                         0B        buildkit.dockerfile.v0
<missing>      29 seconds ago      EXPOSE map[5000/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      29 seconds ago      COPY --chmod=node:node . . # buildkit           11.6MB    buildkit.dockerfile.v0
<missing>      About an hour ago   RUN /bin/sh -c npm i --production # buildkit    16.5MB    buildkit.dockerfile.v0
<missing>      About an hour ago   COPY --chown=node:node package*.json ./ # bu…   185kB     buildkit.dockerfile.v0
<missing>      About an hour ago   USER node                                       0B        buildkit.dockerfile.v0
<missing>      About an hour ago   RUN /bin/sh -c chown node:node /app # buildk…   0B        buildkit.dockerfile.v0
<missing>      2 hours ago         WORKDIR /app                                    0B        buildkit.dockerfile.v0
<missing>      12 days ago         /bin/sh -c #(nop)  CMD ["node"]                 0B
<missing>      12 days ago         /bin/sh -c #(nop)  ENTRYPOINT ["docker-entry…   0B
<missing>      12 days ago         /bin/sh -c #(nop) COPY file:4d192565a7220e13…   388B
<missing>      12 days ago         /bin/sh -c apk add --no-cache --virtual .bui…   5.57MB
<missing>      12 days ago         /bin/sh -c #(nop)  ENV YARN_VERSION=1.22.19     0B
<missing>      12 days ago         /bin/sh -c addgroup -g 1000 node     && addu…   120MB
<missing>      12 days ago         /bin/sh -c #(nop)  ENV NODE_VERSION=20.13.1     0B
<missing>      3 months ago        /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      3 months ago        /bin/sh -c #(nop) ADD file:37a76ec18f9887751…   7.38MB
PS D:\VDT_2024\Midterm\api>
```

Ảnh 10: API Dockerfile history

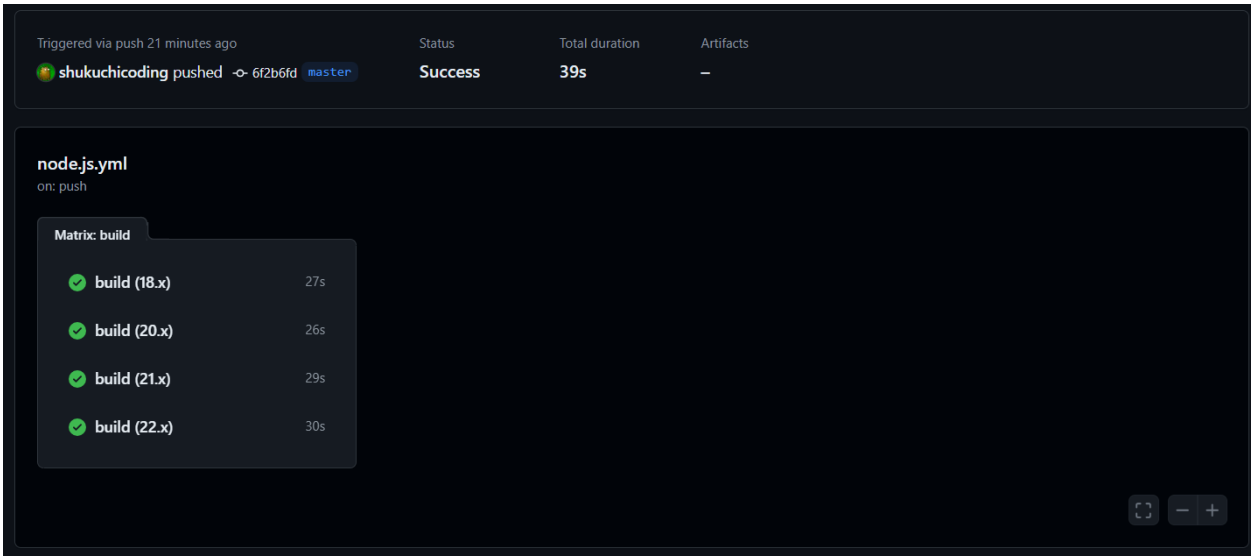
```
PS D:\VDT_2024\Midterm\web> docker build -t vdt-web:1.0.0 .
[+] Building 0.0s (0/0)  docker:default
2024/05/22 23:42:43 http2: server: error reading preface from client //./pipe/docker_engine: file has already been close
[+] Building 3.4s (18/18) FINISHED                                                                          docker:default
 => [internal] load build definition from Dockerfile                                                                0.1s
 => => transferring dockerfile: 527B                                                                                0.0s
 => [internal] load metadata for docker.io/library/node:lts-alpine                                                  2.9s
 => [internal] load metadata for docker.io/library/nginx:stable-alpine                                              2.9s
 => [auth] library/node:pull token for registry-1.docker.io                                                         0.0s
 => [auth] library/nginx:pull token for registry-1.docker.io                                                        0.0s
 => [internal] load .dockerignore                                                                                   0.0s
 => => transferring context: 104B                                                                                   0.0s
 => [stage-1 1/3] FROM docker.io/library/nginx:stable-alpine@sha256:ef587d1eb99e991291c582bfb74f27db27f7ca2c095d4   0.0s
 => [builder 1/7] FROM docker.io/library/node:lts-alpine@sha256:291e84d956f1aff38454bbd3da38941461ad569a185c20aa2   0.0s
 => [internal] load build context                                                                                   0.1s
 => => transferring context: 855B                                                                                   0.0s
 => CACHED [builder 2/7] WORKDIR /app                                                                               0.0s
 => CACHED [builder 3/7] RUN chown node:node /app                                                                   0.0s
 => CACHED [builder 4/7] COPY --chown=node:node package*.json ./                                                    0.0s
 => CACHED [builder 5/7] RUN npm install                                                                            0.0s
 => CACHED [builder 6/7] COPY --chmod=node:node . .                                                                 0.0s
 => CACHED [builder 7/7] RUN npm run build                                                                          0.0s
 => CACHED [stage-1 2/3] COPY --chmod=node:node --from=builder /app/build /usr/share/nginx/html                     0.0s
 => CACHED [stage-1 3/3] COPY --chmod=node:node --from=builder /app/nginx/nginx.conf /etc/nginx/conf.d/default.co   0.0s
 => exporting to image                                                                                              0.1s
 => => exporting layers                                                                                             0.0s
 => => writing image sha256:f50304aa3bbdfe8f1ce6d583d3742948dc6af9f4c791621a14c5eefa67ad1f74                        0.0s
 => => naming to docker.io/library/vdt-web:1.0.0                                                                    0.0s

View build details: docker-desktop://dashboard/build/default/default/y8j6w41rqd5gkljislldn8txg
```

Ảnh 11: Output câu lệnh build Web Dockerfile

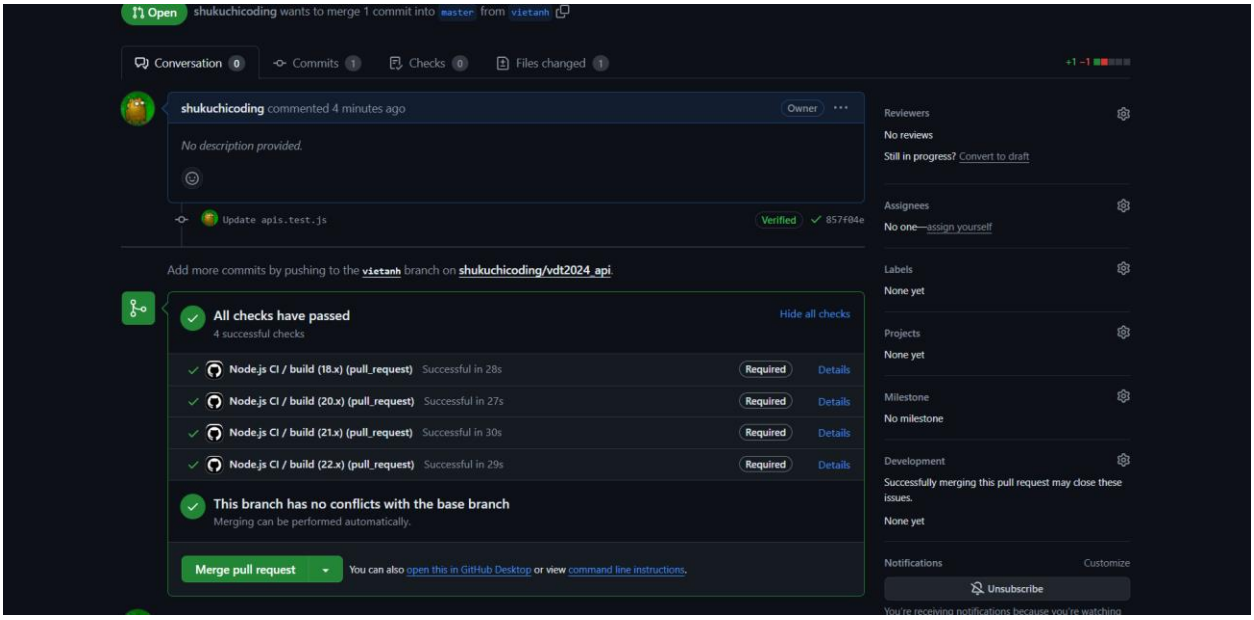
```
View build details: docker-desktop://dashboard/build/default/default/y8j6w41rqd5gkljislldn8txg

What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS D:\VDT_2024\Midterm\web> docker history vdt-web:1.0.0
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
f50304aa3bbd   23 minutes ago   CMD ["nginx" "-g" "daemon off;"]                0B        buildkit.dockerfile.v0
<missing>      23 minutes ago   EXPOSE map[3000/tcp:{}]                         0B        buildkit.dockerfile.v0
<missing>      23 minutes ago   COPY --chmod=node:node /app/nginx/nginx.conf…   177B      buildkit.dockerfile.v0
<missing>      28 minutes ago   COPY --chmod=node:node /app/build /usr/share…   1.97MB    buildkit.dockerfile.v0
<missing>      2 weeks ago      RUN /bin/sh -c set -x     && apkArch="$(cat …   31.2MB    buildkit.dockerfile.v0
<missing>      2 weeks ago      ENV NJS_RELEASE=2                               0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      ENV NJS_VERSION=0.8.4                           0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      CMD ["nginx" "-g" "daemon off;"]                0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      STOPSIGNAL SIGQUIT                              0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      EXPOSE map[80/tcp:{}]                           0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      ENTRYPOINT ["/docker-entrypoint.sh"]            0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      COPY 30-tune-worker-processes.sh /docker-ent…   4.62kB    buildkit.dockerfile.v0
<missing>      2 weeks ago      COPY 20-envsubst-on-templates.sh /docker-ent…   3.02kB    buildkit.dockerfile.v0
<missing>      2 weeks ago      COPY 15-local-resolvers.envsh /docker-entryp…   336B      buildkit.dockerfile.v0
<missing>      2 weeks ago      COPY 10-listen-on-ipv6-by-default.sh /docker…   2.12kB    buildkit.dockerfile.v0
<missing>      2 weeks ago      COPY docker-entrypoint.sh / # buildkit          1.62kB    buildkit.dockerfile.v0
<missing>      2 weeks ago      RUN /bin/sh -c set -x     && addgroup -g 101…   9.68MB    buildkit.dockerfile.v0
<missing>      2 weeks ago      ENV PKG_RELEASE=1                               0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      ENV NGINX_VERSION=1.26.0                        0B        buildkit.dockerfile.v0
<missing>      2 weeks ago      LABEL maintainer=NGINX Docker Maintainers <d…   0B        buildkit.dockerfile.v0
<missing>      3 months ago     /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      3 months ago     /bin/sh -c #(nop) ADD file:37a76ec18f9887751…   7.38MB
PS D:\VDT_2024\Midterm\web>
```

Ảnh 12: Web Dockerfile history

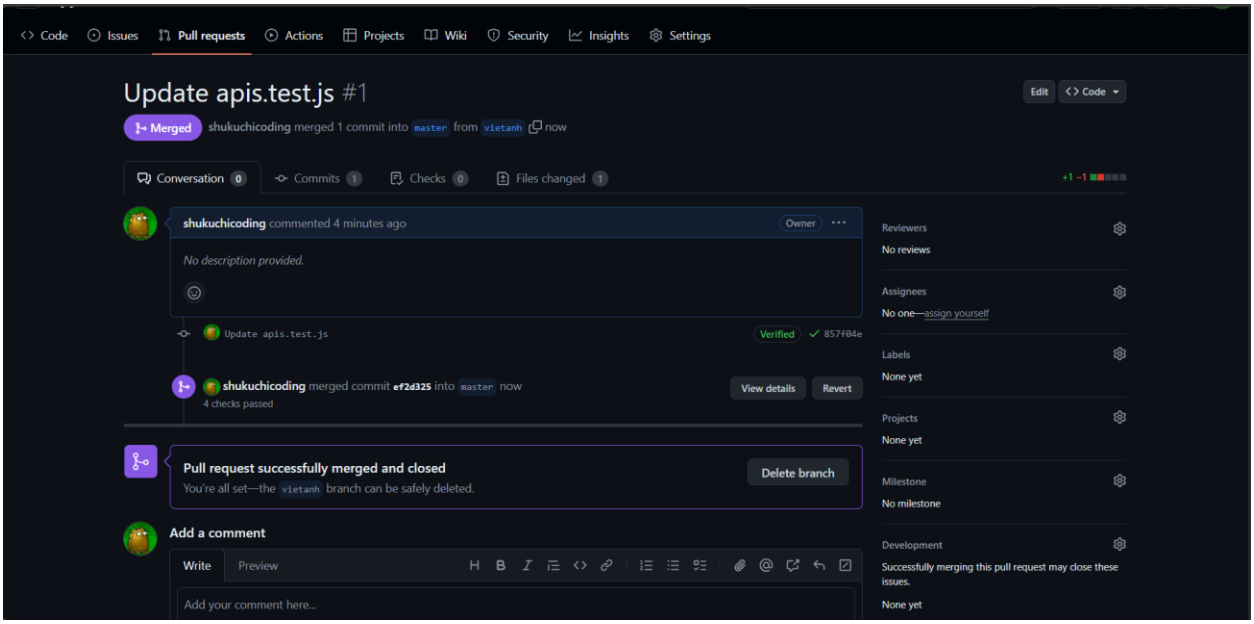
Continuous Integration



Ảnh 13: Output 1 - CI

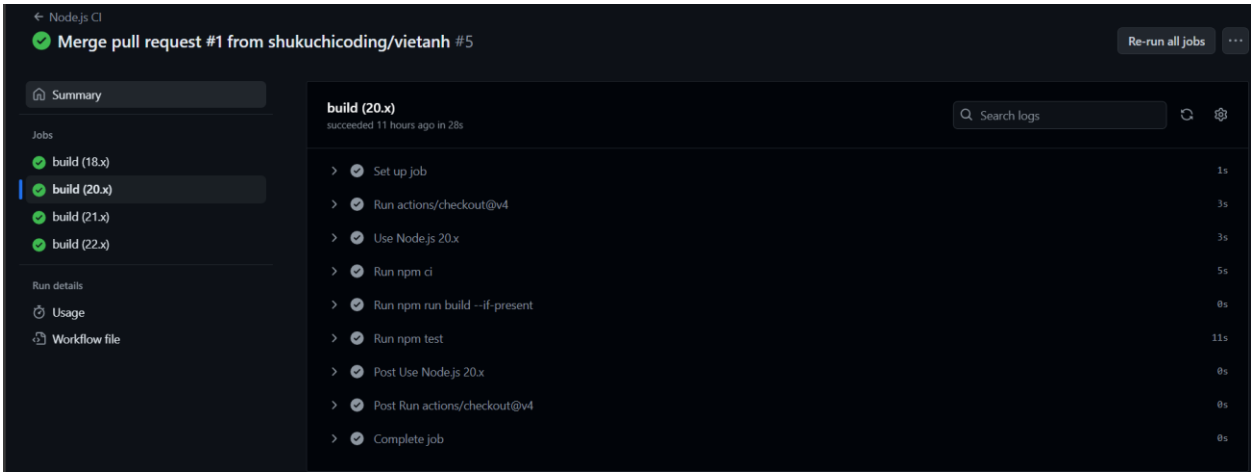


Ảnh 14: Output 2

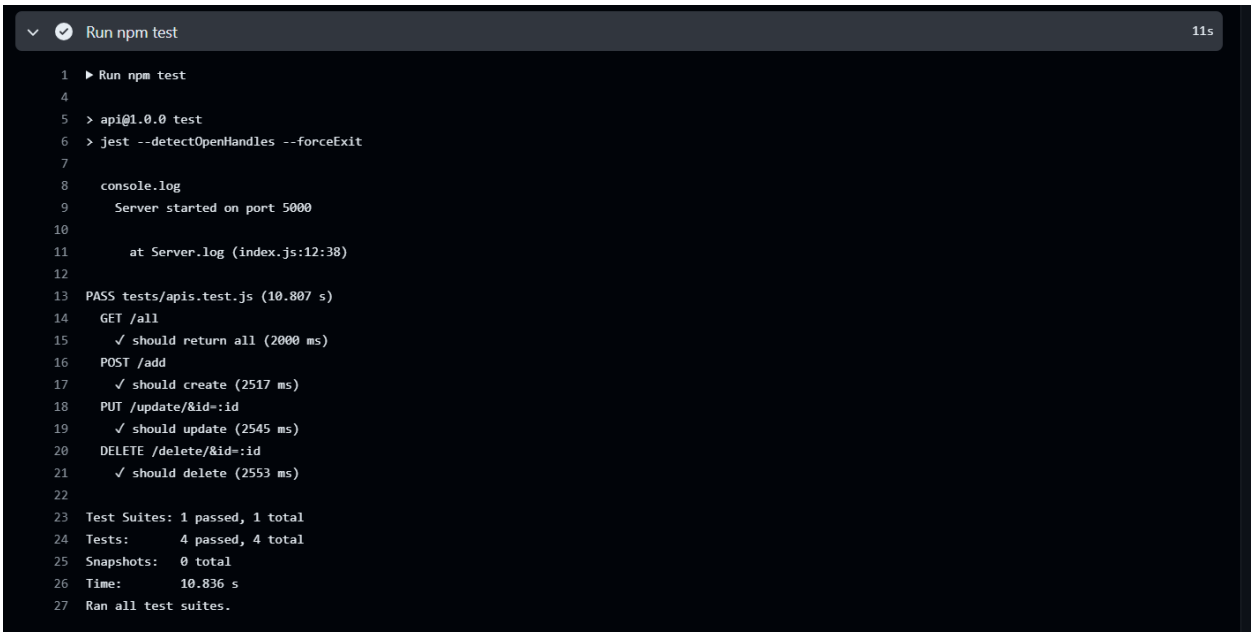


Ảnh 15: Output 3

Khi tạo thêm một nhánh "vietanh" trên repo api, sau khi người dùng nhánh "vietanh" tạo pull request, để merge vào nhánh merge, người quản trị sử dụng công cụ Github Action để tạo luồng tự động tích hợp, chạy các test cases để đảm bảo không có xung đột giữa các phiên bản. Sau quá trình đó, code được merge vào nhánh main, như kết quả ở ảnh 15.

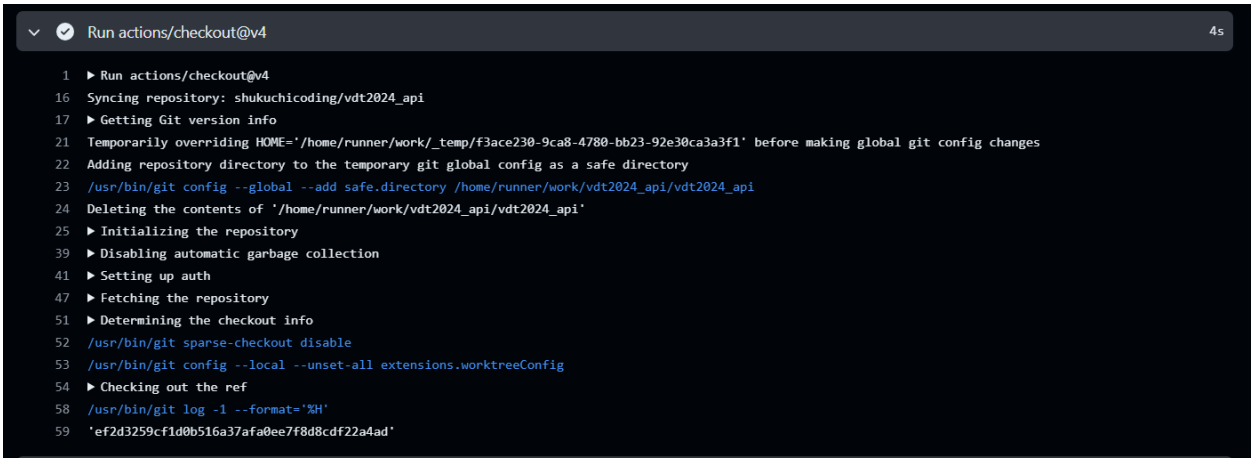


Ảnh 16: Kết quả CI sau khi merge nhánh "vietanh" vào "master" của phiên bản node 20.x



Ảnh 17: Tự động chạy test cases

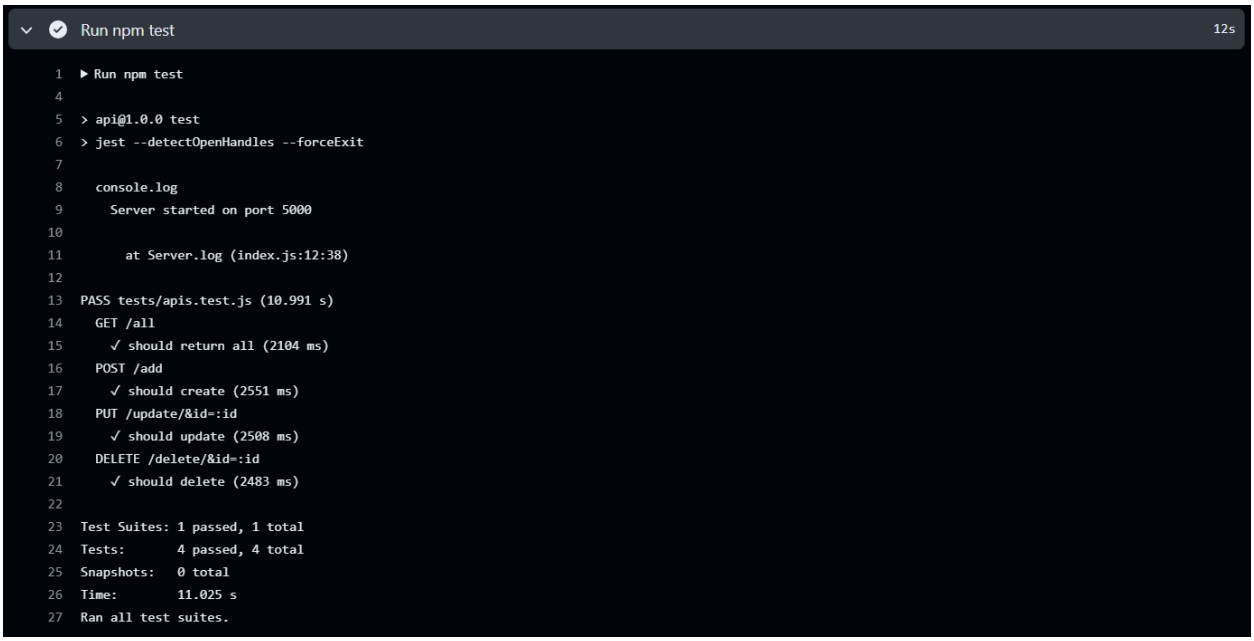
Ảnh 18: CI Log 18.x (1)



Ảnh 19: CI Log 18.x (2)

Ảnh 20: CI Log 18.x (3)

Ảnh 21: CI Log 18.x (4)



Ảnh 22: CI Log 18.x (5)

Ảnh 23: CI Log 18.x (6)

Automation

Link github ansible: https://github.com/shukuchicoding/vdt_ansible.git

Sử dụng 3 máy server cho 3 dịch vụ khác nhau, và 1 server phân phối

ansible > roles > api > ! main.yml

```yaml
---
- name: Deploy API Service
  hosts: api_servers
  become: true
  tasks:
    - name: Pull API Service Docker image
      docker_image:
        name: "{{ api_image }}"
        source: pull

    - name: Run API Service container
      docker_container:
        name: my-api-service
        image: "{{ api_image }}"
        ports:
          - "{{ api_port }}:8080"
        state: started
        restart_policy: always
```

Ảnh 24: API

ansible > roles > db > ! main.yml

```yaml
---
- name: Deploy DB Service
  hosts: db_servers
  become: true
  tasks:
    - name: Pull DB Service Docker image
      docker_image:
        name: "{{ db_image }}"
        source: pull

    - name: Run DB Service container
      docker_container:
        name: my-db-service
      image: "{{ db_image }}"
      ports:
        - "{{ db_port }}:5432"
      state: started
      restart_policy: always
```

Ảnh 25: Database

ansible > roles > web > ! main.yml

```yaml
---
- name: Deploy Web Service
  hosts: web_servers
  become: true
  tasks:
    - name: Pull Web Service Docker image
      docker_image:
        name: "{{ web_image }}"
        source: pull

    - name: Run Web Service container
      docker_container:
        name: my-web-service
        image: "{{ web_image }}"
        ports:
          - "{{ web_port }}:80"
        state: started
        restart_policy: always
```

Ảnh 26: Web

Các biến được sử dụng để có thể thay đổi: web_image, api_image, db_image. Ở đây, web_image = web, api_image = api, db_image = mongo:latest, web_port = 3000, api_port = 5000, db_port = 27017.

ansible > inventory.ini

```
[web_servers]
web1 ansible_host=192.168.1.10 web_port=3000 web_image=web:1.0.0
web2 ansible_host=192.168.1.11 web_port=3000 web_image=web:1.0.0

[api_servers]
api1 ansible_host=192.168.1.20 api_port=5000 api_image=api:1.0.0
api2 ansible_host=192.168.1.21 api_port=5000 api_image=api:1.0.0

[db_servers]
db1 ansible_host=192.168.1.30 db_port=27017 db_imgae=mongo:latest
```

Ảnh 27: Inventory

ansible > playbook.yml

```
---
- name: Deploy Web Service
  hosts: web_servers
  become: true
  roles:
    - web

- name: Deploy API Service
  hosts: api_servers
  become: true
  roles:
    - api

- name: Deploy DB Service
  hosts: db_servers
  become: true
  roles:
    - db
```

Ảnh 28: Playbook

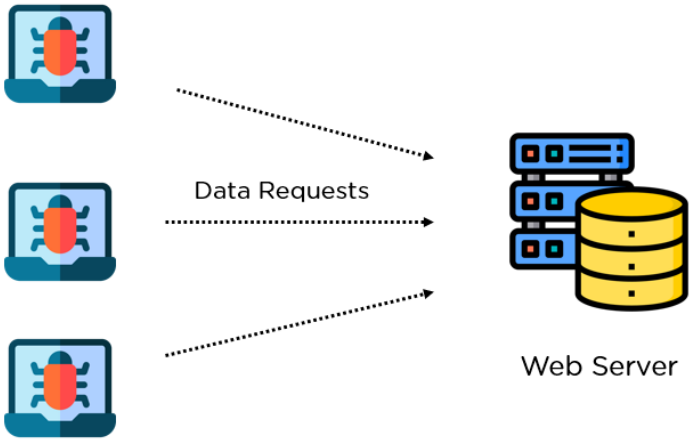
Phần 2: Nghiên cứu

Tổng quan về cân bằng tải

Nguyên nhân cần có cân bằng tải

Trong kiến trúc Client - Server, một server được kết nối đến nhiều client khác nhau, các thông tin trao đổi là các thông điệp Yêu cầu - Phản hồi. Trong một hệ thống lớn, một server phải tiếp nhận nhiều yêu cầu từ phía client, ví dụ như hệ thống phân giải tên miền (DNS), các hệ thống bán hàng trực tuyến, … Điều này gây ra quá tải đường truyền, nghẽn mạng, …

Trước tiên, xem xét một kịch bản sau: Một ứng dụng chạy trên một server duy nhất và clients truy cập trực tiếp tới server.



Có hai vấn đề với mô hình trên:

Đầu tiên, điểm đơn chịu lỗi: Việc chỉ sử dụng một server làm giảm tính đáp ứng của hệ thống. Nếu một server bị lỗi, các request từ clients sẽ bị gián đoạn, làm giảm trải nghiệm người dùng.

Thứ hai, quá tải server: Một server chỉ có thể kiểm soát hữu hạn các request từ clients, do đó sẽ rất khó khăn để mở rộng hệ thống.

Vì vậy, giải pháp được đưa ra: cân bằng tải.

Định nghĩa

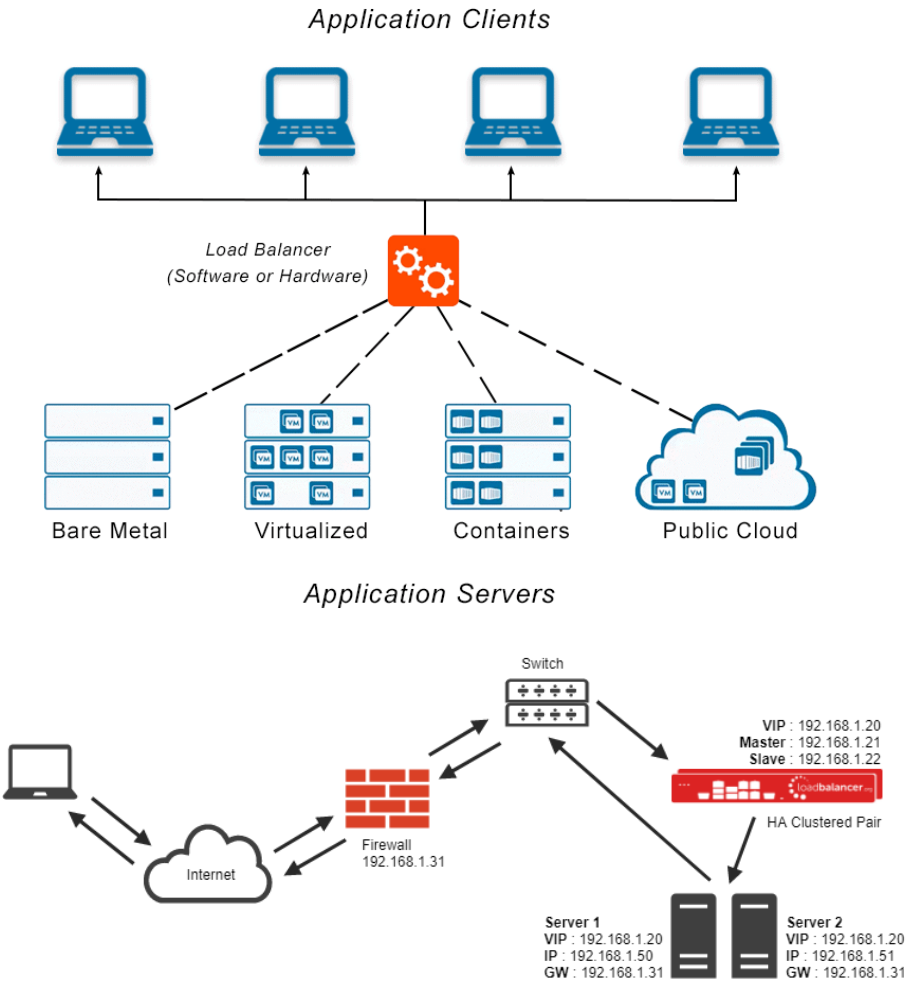
Cân bằng tải là quá trình phân phối một tập hợp các tác vụ trên một tập hợp tài nguyên tính toán, với mục tiêu làm tăng hiệu quả, tính đáp ứng của hệ thống.

Bộ cân bằng tải là một thiết bị hoặc phần mềm giữa clients và servers, đảm bảo tất cả tài nguyên có hiệu quả sử dụng như nhau.

Nguyên lý hoạt động

Cân bằng tải có thể được triển khai bằng phần cứng bởi các thiết bị phần cứng hoặc phần mềm được tích hợp trên các máy chủ hoặc được phân phối dưới dạng dịch vụ đám mây.

Trong cả hai trường hợp, cân bằng tải hoạt động bằng cách sắp xếp các yêu cầu của clients trong thời gian thực và xác định server nào có khả năng phục vụ tốt nhất. Bộ cân bằng tải sẽ định tuyến các requests đến bất kỳ server có khả năng đáp ứng.



1. Phân phối requests
   Khi một request được client gửi đến server, như yêu cầu truy cập tài nguyên, request được đưa đến bộ cân bằng tải. Bộ cân bằng tải sẽ chọn backend servers, các backend servers có thể ở dạng vật lý, ảo hóa, containers, …

2. Health checking
   Health check các server, xem server nào có khả năng đáp ứng request, một server bị sập sẽ được bộ cân bằng tải bỏ qua.
3. Load balancing algorithm
   Bộ cân bằng tải sử dụng thuật toán để điều phối lưu lượng các request.
4. Session persistence
   Trong một số trường hợp yêu cầu tính liên tục của phiên làm việc, hoặc do tài nguyên được lưu trữ cục bộ trên một server nhất định, bộ cân bằng tải được cấu hình để request được ứng đến nhất quán một server, duy trì phiên.
5. Scalability and high availability
   Bộ cân bằng tải đóng vai trò mở rộng hệ thống theo chiều ngang. Khi lưu lượng tăng, nó sẽ chọn thêm server để đáp ứng và giảm bớt server trong trường hợp còn lại.
6. SSL Termination
   Bộ cân bằng tải hỗ trợ mã hóa/giải mã SSL thay các server, cải thiện hiệu suất tổng thể của hệ thống.

Hai cách triển khai cân bằng tải

Có hai cách triển khai cân bằng tải, đó là bằng phần cứng và phần mềm. Dưới đây là bảng so sánh hai cách cách triển khai này.

Đặc điểm	Software Load Balancers	Hardware Load Balancers
Hình thức triển khai	Bằng phần mềm cài trên các servers, dễ dàng	Bằng các thiết bị vật lý, được thiết kế cho mục đích duy nhất là cân bằng tải
Giá thành	Rẻ hơn phần cứng	Đắt hơn phần mềm
Khả năng mở rộng	Dễ dàng mở rộng theo chiều ngang bằng cách thêm các máy ảo	Mở rộng bằng việc thêm các thiết bị (tốn kém)
Cấu hình	Linh hoạt	Kém linh hoạt
Kiểm soát thông lượng	Có giới hạn nhỏ	Có thể kiểm soát heavy traffics
Usecases	Phù hợp cho các ứng dụng cloud và môi trường ảo hóa	Hiệu quả khi triển khai quy mô lớn, yêu cầu bảo mật nghiêm ngặt
Ví dụ	Nginx, HAProxy, …	F5 Networks, Barracuda LoadBalancer, …

Lợi ích của cân bằng tải

Tính sẵn sàng (availability):

Bộ cân bằng tải thực hiện healthcheck trên các servers trước khi định tuyến các requests đến chúng. Nếu một server bị lỗi, hoặc cần bảo trì, nâng cấp, bộ cân bằng tải sẽ tự động định tuyến requests sang các servers khác, duy trì tính sẵn sàng cao (High Availability)

Tính mở rộng (scalability):

Các máy chủ vật lý hoặc máy chủ ảo có thể được lấy thêm hoặc xóa bớt đi khi cần thiết, làm hệ thống dễ mở rộng hơn.

Tính bảo mật (security):

Bộ cân bằng tải có thể bao gồm các tính năng bảo mật như SSL, WebApp Firewall (WAF), xác thực đa yếu tố (Multi-factor Authentication). Nó cũng có thể được tích hợp vào App Delivery Controller (ADC). Với việc tăng hoặc giảm lưu lượng truy cập một cách an toàn, cân bằng tải giúp giảm rủi ro bảo mật, ví dụ tấn công DDoS trên hệ phân tán.

Quy trình xây dựng một bộ cân bằng tải

1. Chọn cách triển khai cân bằng tải (bằng phần cứng hoặc phần mềm)
2. Thiết kế network cho các thiết bị sử dụng cân bằng tải
3. Cài đặt và cấu hình bộ cân bằng tải (địa chỉ IP, cổng kết nối, thông tin backend server)
4. Chọn thuật toán cân bằng tải
5. Cấu hình Health Checks để giám sát trạng thái backend
6. Cấu hình Persistence (nếu cần)
   Khi một yêu cầu từ một khách hàng đến load balancer, thông thường load balancer sẽ định tuyến yêu cầu đó đến một máy chủ backend có sẵn và đang có hiệu suất tốt nhất. Tuy nhiên, với session persistence được kích hoạt, load balancer sẽ lưu trữ thông tin nhận dạng duy nhất (như địa chỉ IP của khách hàng) và đảm bảo rằng các yêu cầu tiếp theo từ cùng một khách hàng sẽ luôn được định tuyến đến cùng một máy chủ backend.
7. Kiểm tra, đảm bảo các requests đến được đúng servers
8. Mở rộng khi cần thiết
9. Xem xét tính bảo mật
10. Giám sát và bảo trì

Cloud load balancing

Cloud Load Balancing là một dịch vụ trong môi trường điện toán đám mây, cung cấp khả năng cân bằng tải tự động cho ứng dụng và dịch vụ trên các nguồn tài nguyên phân tán, được cung cấp bởi các nhà cung cấp đám mây như Amazon Web Services (AWS), Microsoft Azure và Google Cloud Platform. Trên cloud, mỗi Server được coi như một máy ảo (VM).

Mục này được trình bày dựa trên backend của Google Cloud.

Cloud load balancing hỗ trợ các tính năng:

- Single anycast IP address: cung cấp cân bằng tải liên khu vực (cơ chế giống với CDN hay DNS, chọn server gần người dùng nhất để phản hồi)
- Seamless autoscaling: Mở rộng linh hoạt
- Software-defined load balancing
- Layer 4 and layer 7 load balancing: cân bằng tải ở tầng giao vận (định tuyến dựa vào giao thức UDP/TCP, ICMP, … và tầng ứng dụng trong mô hình OSI (định tuyến dựa vào HTTP header, …)
- External and internal load balancing: cân bằng tải cho các kết nối từ ngoài vào môi trường Google cloud, và cân bằng bên trong môi trường Google cloud
- Global and regional load balancing: phân phối tài nguyên cân bằng tải trên nhiều khu vực, định tuyến kết nối đến các máy chủ gần nhất để phản hồi, giảm độ trễ và tăng tính khả dụng của hệ thống
- IPv6, VIPs: Định tuyến cho cả dải địa chỉ IPv6 và mạng riêng ảo

Các thuật toán cân bằng tải trong cloud computing

Phân loại thuật toán cân bằng tải truyền thống

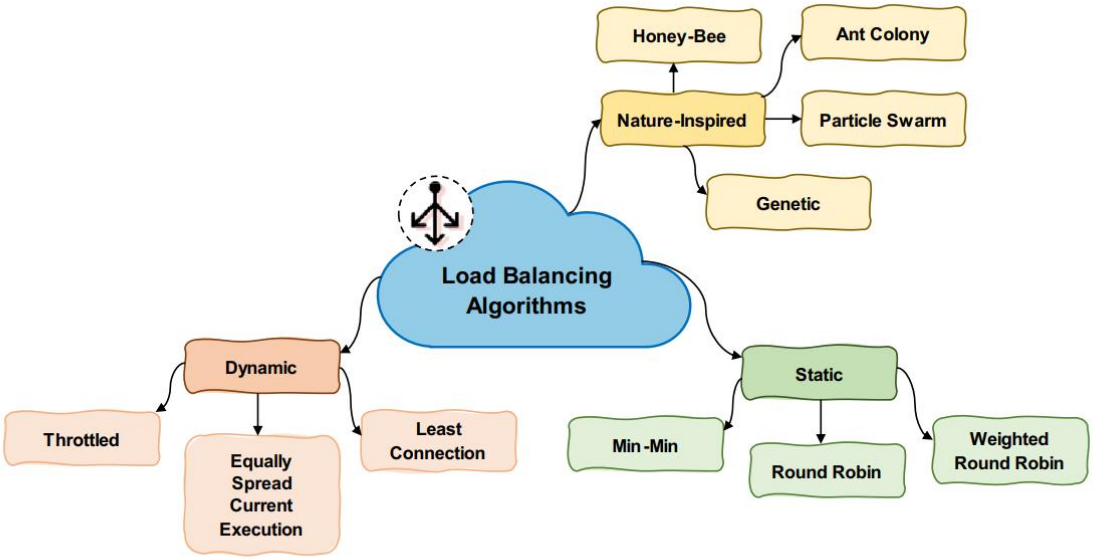


Fig. 7. Taxonomy of Load Balancing Algorithms.

Cân bằng tải truyền thống thường được chia thành 2 loại: Cân bằng tĩnh và cân bằng động.

Cân bằng tải tĩnh (Static load balancing algorithms) là một phương pháp cân bằng tải trong hệ thống mạng mà việc phân phối lưu lượng truy cập đến các máy chủ backend được cấu hình một cách tĩnh và không thay đổi theo thời gian hoặc điều kiện khác.

Trong static load balancing, một nguyên tắc đơn giản được áp dụng để phân phối lưu lượng truy cập: các yêu cầu từ người dùng được chuyển hướng đến các máy chủ backend theo một tiêu chí cố định. Ví dụ, lưu lượng truy cập có thể được phân phối theo thứ tự các máy chủ trong danh sách cấu hình, hoặc dựa trên một thuật toán cụ thể như round-robin (lần lượt), hash-based (dựa trên hash), hoặc weighted (có trọng số).

Ưu điểm của static load balancing là đơn giản và dễ triển khai. Nó không đòi hỏi sự phức tạp trong việc giám sát và điều chỉnh cân bằng tải trong thời gian thực. Tuy nhiên, phương pháp này có nhược điểm là không linh hoạt trong việc thích ứng với sự thay đổi trong môi trường mạng. Khi một máy chủ backend trở nên quá tải hoặc không khả dụng, static load balancing không có khả năng tự động chuyển hướng lưu lượng truy cập đến các máy chủ khác.

Trong các môi trường mạng đơn giản hoặc với một số yêu cầu cụ thể, static load balancing có thể là một giải pháp hợp lý. Tuy nhiên, trong các hệ thống có quy mô lớn hoặc yêu cầu tính sẵn sàng cao, phương pháp cân bằng tải động (dynamic load balancing) thường được ưu tiên do khả năng linh hoạt và tự động hóa trong việc điều chỉnh và phản hồi lại sự thay đổi trong môi trường mạng.

Cân bằng tải động (dynamic load balancing algorithms) là một phương pháp cân bằng tải trong hệ thống mạng mà việc phân phối lưu lượng truy cập đến các máy chủ backend được điều chỉnh và thay đổi động dựa trên các yếu tố như tình trạng máy chủ, tải hiện tại, độ trễ mạng, hoặc các yếu tố khác.

Trong dynamic load balancing, một hệ thống quản lý cân bằng tải (load balancer) thông minh được sử dụng để giám sát và phân phối lưu lượng truy cập dựa trên các thông tin thời gian thực. Các quyết định về phân phối lưu lượng có thể được đưa ra dựa trên thuật toán động như Least Connections (ít kết nối nhất), Response Time (thời gian phản hồi), hoặc Adaptive (thích ứng). Các thông số này được tính toán và cập nhật liên tục để đảm bảo rằng lưu lượng truy cập được chuyển đến các backend khả dụng và có hiệu suất tốt nhất.

Một số lợi ích của dynamic load balancing bao gồm:

Tự động phản ứng với sự thay đổi: Dynamic load balancing có khả năng phản ứng tức thì với sự thay đổi trong tình trạng máy chủ backend, lưu lượng truy cập hoặc mạng. Nó có thể điều chỉnh phân phối lưu lượng để đáp ứng yêu cầu và tối ưu hóa hiệu suất hệ thống.

Tăng tính sẵn sàng: Với dynamic load balancing, nếu một máy chủ backend trở nên quá tải hoặc không khả dụng, lưu lượng truy cập sẽ tự động được chuyển hướng đến các máy chủ khác đang hoạt động tốt. Điều này cải thiện tính sẵn sàng và đảm bảo rằng dịch vụ vẫn hoạt động mà không gây gián đoạn cho người dùng.

Tối ưu hóa hiệu suất: Dynamic load balancing có thể điều chỉnh phân phối lưu lượng để tận dụng tối đa tài nguyên và khả năng xử lý của các backend. Nó giúp đảm bảo rằng mỗi máy chủ backend được sử dụng một cách hiệu quả và giảm tải không đồng đều trên hệ thống.

Tích hợp với các hệ thống khác: Dynamic load balancing có thể tích hợp với các công nghệ và dịch vụ khác trong hệ thống mạng như cloud computing, hệ thống phân tán và ảo hóa để cung cấp tính linh hoạt và mở rộng.

Các thuật toán lấy cảm hứng từ tự nhiên (Nature-inspired Load Balancing (NLB) Algorithms) là những thuật toán cân bằng tải được lấy cảm hứng từ tự nhiên và các quy tắc hoạt động trong tự nhiên. Các thuật toán này sử dụng các khái niệm và cơ chế từ tự nhiên để tối ưu hóa quá trình cân bằng tải trong hệ thống mạng.

Các NLB Algorithms thường dựa trên các nguyên lý hoạt động của các hệ thống tự nhiên như kiến trúc đàn tự nhiên, quy hoạch di cư của đàn chim, hiệu suất câu mồi của cá sấu, sự phân bố thực phẩm trong hệ sinh thái, và nhiều khía cạnh khác của tự nhiên.

Một số ví dụ về NLB Algorithms bao gồm:

Ant Colony Optimization (ACO): Được lấy cảm hứng từ hoạt động của kiến, thuật toán ACO sử dụng quy hoạch của kiến để tìm kiếm và cân bằng tải các tài nguyên.

Particle Swarm Optimization (PSO): Lấy cảm hứng từ cách đàn chim tìm kiếm thức ăn, thuật toán PSO sử dụng các phần tử (particle) để tìm kiếm và điều chỉnh cân bằng tải trong hệ thống.

Genetic Algorithm (GA): Sử dụng các khái niệm từ di truyền học, thuật toán GA tạo ra các thế hệ "cá thể" và áp dụng các quy tắc di truyền để tối ưu hóa quá trình cân bằng tải.

Firefly Algorithm (FA): Lấy cảm hứng từ cách con đom đóm tương tác với nhau, thuật toán FA sử dụng sự phát sáng của đom đóm để tìm kiếm và tối ưu hóa cân bằng tải.

Các thuật toán cân bằng tĩnh điển hình

Thuật toán Round robin

1. Các máy chủ backend được xếp hàng theo một thứ tự cố định hoặc được quản lý trong danh sách.
2. Khi một yêu cầu đến, load balancer sẽ chuyển tiếp yêu cầu đến máy chủ đầu tiên trong danh sách.
3. Máy chủ nhận yêu cầu và xử lý nó.
4. Sau đó, máy chủ được đưa xuống cuối danh sách, và yêu cầu tiếp theo được chuyển tiếp đến máy chủ tiếp theo trong danh sách.
5. Quá trình này được lặp lại cho mỗi yêu cầu mới, và danh sách máy chủ được duy trì trong trạng thái tuần tự.

Thuật toán dựa trên mã băm

1. Khi một yêu cầu đến, thông tin quan trọng trong yêu cầu (như địa chỉ IP nguồn, URL, session ID, hoặc thông tin tùy chỉnh khác) được sử dụng làm đầu vào cho hàm băm.
2. Hàm băm sẽ ánh xạ thông tin đầu vào thành một giá trị hash duy nhất.
3. Giá trị hash này sẽ được sử dụng để xác định máy chủ backend mà yêu cầu sẽ được chuyển tiếp đến. Ví dụ, giá trị hash có thể được chia lấy dư cho số lượng máy chủ để xác định máy chủ tương ứng.
4. Yêu cầu được chuyển tiếp đến máy chủ được xác định bằng giá trị hash.

Thuật toán Weighted round robin

1. Các máy chủ backend được xếp hàng trong một danh sách và được gán một trọng số tương ứng.
2. Khi một yêu cầu đến, load balancer sẽ chuyển tiếp yêu cầu đến máy chủ có trọng số cao nhất trong danh sách.
3. Máy chủ nhận yêu cầu và xử lý nó.
4. Sau đó, trọng số của máy chủ được giảm đi một lượng xác định hoặc nâng lại trở thành trọng số ban đầu.
5. Quá trình này được lặp lại cho mỗi yêu cầu mới, và yêu cầu sẽ được chuyển tiếp đến máy chủ tiếp theo có trọng số cao nhất.

Đây là một cải tiến của thuật toán Round robin.

Các thuật toán cân bằng động điển hình

Thuật toán Least Connection

Lựa chọn VM có ít kết nối nhất để thực hiện phản hồi request.

Thuật toán Throttled

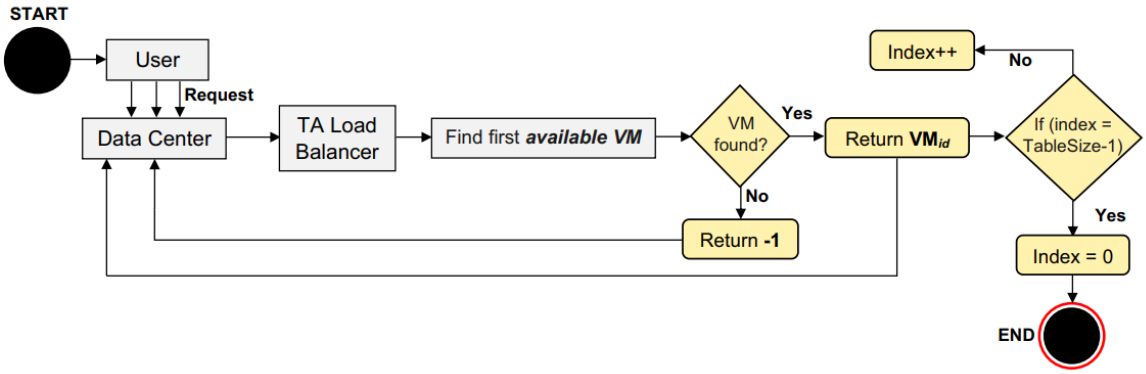


Fig. 9. Revised Throttled Algorithm Flowchart Adopted From (Bhagyalakshmi and D. Malhotra, "A Review, , 2017).

Mỗi VM có trạng thái sẵn sàng, bận, rỗi. Nếu VM sẵn sàng và đủ không gian xử lý, request sẽ được allocated trong VM có id được TA trả về, nếu không tìm thấy VM, TA return -1, hoặc sẽ đưa request vào hàng đợi.

Thuật toán Equally Spread Current Execution

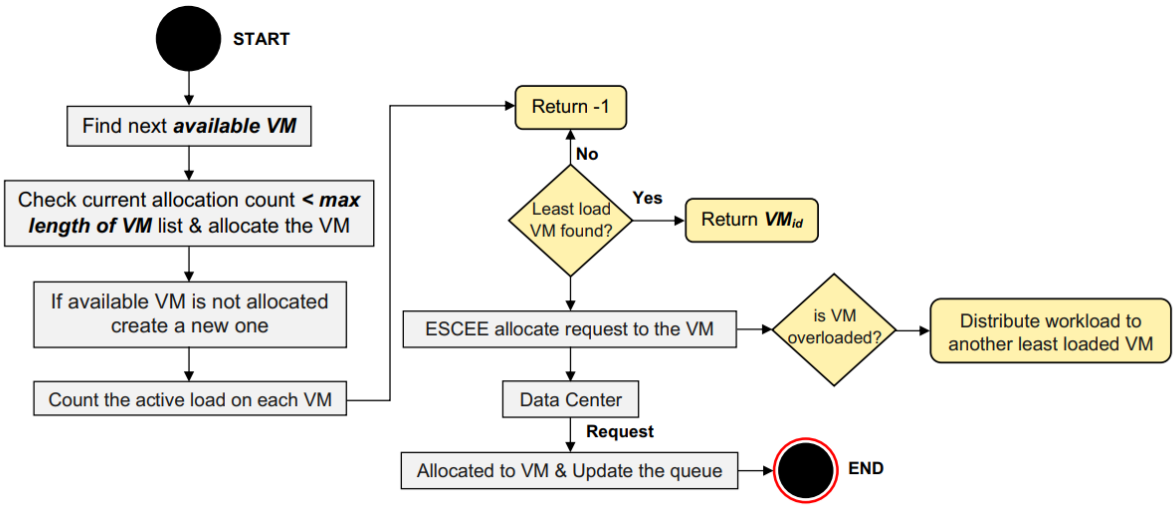


Fig. 10. Equally Spread Current Execution Algorithm Flowchart.

ESCE (Equally Spread Current Execution) là một loại thuật toán động trong cân bằng tải. Nó xem xét kích thước công việc như một ưu tiên, sau đó phân phối công việc cho một máy ảo (VM) có tải nhẹ một cách ngẫu nhiên. Nó cũng được biết đến là kỹ thuật Spread Spectrum vì nó phân tán công việc đến các nút khác nhau. ESCE phụ thuộc vào việc sử dụng hàng đợi để lưu trữ các yêu cầu và phân phối tải cho các VM nếu có VM nào đó bị quá tải. Một vấn đề phổ biến của ESCE là nó có thể gây ra chi phí thêm khi cập nhật bảng chỉ mục do sự giao tiếp giữa bộ điều khiển Trung tâm Dữ liệu và bộ cân bằng tải.

Các thuật toán cân bằng tải lấy cảm hứng từ tự nhiên

Thuật toán Honey Bee

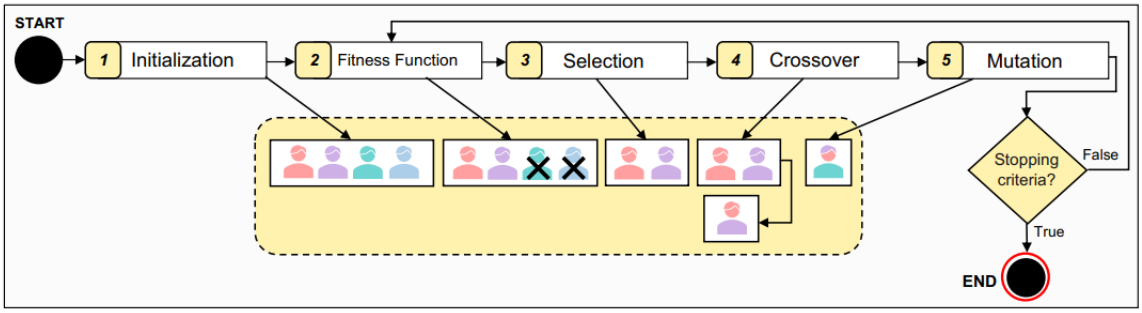


Fig. 14. Genetic Algorithm.

Ý tưởng của thuật toán này là một nhóm ong được gọi là ong săn mồi (foraging bees) tìm kiếm nguồn thức ăn và gửi thông tin vị trí tới các ong khác. Điều này được biết đến để giải quyết các vấn đề quyết định và phân loại một cách linh hoạt và mạnh mẽ hơn. Như có thể thấy trong sơ đồ, sức khỏe của các con ong được đánh giá mỗi lần và quá trình tìm kiếm thức ăn được lặp lại.

Tương tự, trong môi trườngcloud, có nhiều sự thay đổi về yêu cầu trên các máy chủ, và các dịch vụ được phân bổ một cách động và các máy ảo (VM) nên được tận dụng tối đa để giảm thời gian chờ đợi. Hành vi của ong mật có thể được ánh xạ vào môi trường đám mây như được thể hiện trong bảng dưới đây. Vấn đề của thuật toán này là các công việc có ưu tiên thấp sẽ phải chờ đợi trong hàng đợi.

Table 5
How to Map Bee Colony with Cloud Environment (Babu et al., 2015).

Honey-Bee Hive	Cloud Environment
Honey-Bee	Task (Cloudlet)
Food source	Virtual Machine
Searching (foraging) for food source	Task is loaded to the Virtual Machine
Honey-Bee getting crowded (depleted) near the food source	Overloading state of the Virtual Machine
A new food source is found.	Task is removed from overload Virtual Machine and scheduled to another, underloaded Virtual Machine.

Thuật toán Ant Colony Optimization

Thuật toán này dựa trên hành vi của kiến trong quá trình tìm kiếm thức ăn. Kiến di chuyển ngẫu nhiên tìm kiếm thức ăn và khi trở về tổ, chúng phát ra một lượng hóa chất được gọi là mùi kiến (pheromone). Lượng mùi kiến này xác định đường đi ngắn nhất từ tổ đến nguồn thức ăn để cho các kiến khác theo sau. Mặc dù phương pháp này tốt cho tối ưu hóa tài nguyên, tuy nhiên, nó dẫn đến thời gian phản hồi chậm và hiệu suất kém. Hành vi của kiến trong ACO được minh họa trong hình dưới đây.

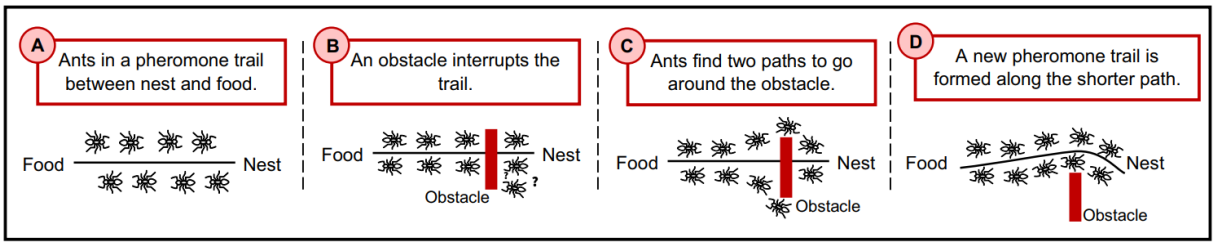


Fig. 16. Revised Environment of Ant Colony Optimization Algorithm Adopted From (Bajpai and Yadav, 2015).

Tài liệu tham khảo

- What is Load Balancer & How Load Balancing works? - GeeksforGeeks
- Cloud load balancing - Wikipedia
- Cloud Load Balancing overview | Google Cloud
- Load balancing in cloud computing – A hierarchical taxonomical classification, Afzal and Kavitha, Journal of Cloud Computing: Advances, Systems and Applications (2019) 8:22, https://doi.org/10.1186/s13677-019-0146-7
- Load balancing techniques in cloud computing environment: a review, J. King Saud Univ, Comput. Inf. Sci., 34 (7) (2022), pp. 3910-3933, https://doi.org/10.1016/j.jksuci.2021.02.007