Bluebik Senior DevOps Engineer Exam

Before taking the exam, *please read the instructions below*:

• Do not use any Generative AI or consult other individuals to answer the questions. If we find any plagiarism, the interview will be "terminated"

.

• Write the answer in a document file and render it in "PDF" format.

• You can answer the questions in English or Thai.

• For inquiries, please contact at punnapop.c@bluebik.com or ekdanai.d@bluebik.com.

1. Scenario: Your current software development and deployment process follows an environment-based branching strategy, where separate branches are maintained for different environments (e.g., dev, sit, uat, main, tag: vx.x.x). The CI/CD pipeline consists of three main stages for any branches:

• Clone Stage

- The pipeline pulls the source code from the corresponding Git branch based on the target environment.
- Developers push changes to the relevant environment branch (dev, sit, uat, tag: vx.x.x).

• Build Stage

- A Docker image is built using the application source code.
- The image is tagged with "latest" and pushed to a container registry (e.g., AWS Elastic Container Registry - ECR).
- The "latest" tag ensures that the most recent image is always deployed.

• Deploy Stage

- The pipeline updates the ECS Fargate service with the newly built "latest" Docker image.
- The service automatically pulls the "latest" image from the registry and replaces the currently running tasks

Please answer the following:

*** การตอบคำถามข้อสอบต่อไปนี้จะใช้ Keyword ที่เป็นชื่อย่อของ AWS เป็นหลักครับ

1. Is this approach following best practices? Why?

   Solve: ส่วนตัวคิดว่าแนวนี้ทางนี้ยังไม่ใช่แนวทางที่ดีที่สุด เท่าที่เคยทำมาอาจจะเจอปัญหาต่างๆลองแยกเป็นข้อๆ ดังนี้

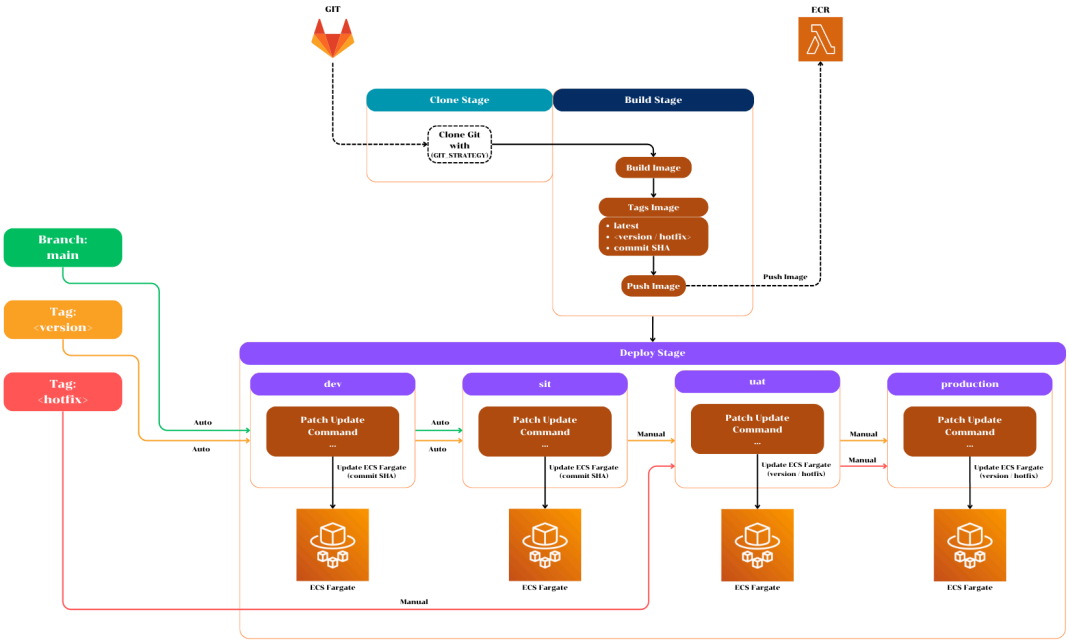
   - Clone Stage: การทำกฎ Pipeline แยก Branch ทั้งหมดแบบนี้ แปลว่า Code จะถูก Run Pipeline แยกกันทั้งหมดตาม Environment ซึ่งมันจะขึ้นอยู่กับนักพัฒนาเลยว่าทำเอาโค้ด Commit ไหนขึ้นมาบน Branch ไหน ปัญหาคือ Version Control อาจจะผิดไม่เป็นไปตาม Flow (Developer Debt) อาจเกิดปัญหา ดังนี้
     1. (Develop Risk) ลองคิดภาพว่าแต่ละ Branch อาจมีโค้ดที่แตกต่างกัน เช่น
        a. อาจมี hotfix ใน uat แต่ไม่มีใน dev
        b. อาจมีแก้ไข Bug feature ใน dev แต่ไม่ได้ deploy ไปที่ sit
        c. แก้ไขโค้ดโดยตรงที่ sit โดยไม่ผ่าน dev ก่อน (ไม่ควรทำแต่เกิดขึ้นได้จริงๆ) เพราะต้องการความรวดเร็วในการส่งขึ้นให้ Tester/QA ทดสอบ
     2. (Conflict Overhead) ต้องมีการ Merge Code ไปๆมาๆเสี่ยงต่อการเจอ Conflict (อาจมีการทำ Merge Request และอาจจะมีคน Merge เป็น Lead แค่ไม่กี่คนเพื่อแก้ปัญหา Conflict พวกนี้ แต่จริงๆถ้าทำ Stage นี้ดีๆจะลดงานคนเหล่านั้นได้)

3. (Testing Confidence) -> การ Testing จะยากเพราะ Tester/QA (แม้แต่นักพัฒนาเอง) ไม่มีความมั่นใจเลยว่าโค้ดแต่ละ Environment มีการ Control Version ผิดพลาดหรือไม่
4. (Pipeline Flow) -> Pipeline ก็จัดการยากเพราะ Job แต่ละ Environment แยกกันหมดไม่มีความต่อเนื่องกันของ Pipeline Flow

- Build Stage: ขั้นตอนการ Build นี้การ Push Tag latest ไม่ได้ผิด แต่ไม่ควรนำ latest มาใช้ในการ Control deploy แต่ละ Environment เพราะอาจเกิดปัญหา ดังนี้
    1. Tag latest ไม่ได้หมายถึงล่าสุดเสมอ เป็นแค่ Tag ธรรมดาๆที่นิยมใช้กัน
    2. ปัญหาที่ตามมาคือถ้ามีการ Run Pipeline หรือวิธีการใดๆที่อาจทำให้ deploy ซ้ำซ้อนกัน Tag latest นั้นอาจถูกเขียนทับระหว่างทาง และทำให้การสั่ง Update ECS Fargate ใช้ชุดโค้ดผิดจากที่ต้องการไว้
    3. ทำ Rollback ไม่ได้หาปัญหาก็ยากเพราะไม่รู้ว่าตอนนี้ Tag latest คือ Commit ไหนกันแน่ (การ Push Commit ที่ไม่มีปัญหาขึ้นไปใหม่เพื่อให้มันเป็น latest และไปแทนที่โค้ดที่มีปัญหา ก็อาจจะทำได้แต่ก็เสียเวลาไปอีก)
    4. การเฝ้าติดตามฝั่ง ECS Fargate แต่ละ Environment ก็ยากเพราะไม่รู้เลยว่าตอนนี้ใช้ Image ที่ Commit หรือ Version ไหน เพราะทุกอันเป็น Tag Latest
- Deploy Stage: ต่อเนื่องจาก Build Stage เลยคือจะเจอปัญหาเรื่องการใช้ Tag latest ซึ่งพฤติกรรมของ ECS Fargate ไม่ได้จะดึง ECR มาโดยอัตโนมัติ มันไม่รู้ด้วยซ้ำว่ามี Image ใหม่เข้ามา แล้วถ้าใช้ Tag latest อาจเกิดปัญหา ดังนี้
    1. ปัญหาต่างๆจาก Build Stage ที่กล่าวมา
    2. ECS Fargate จะไม่ได้ดึง Image ล่าสุดเสมอ ซึ่งถ้าตามปกติอาจมีการ Cache ตาม Tag ไว้ด้วยทำให้ไม่ได้ Image ล่าสุดเพราะดันใช้ latest เหมือนกัน
    3. การใช้คำสั่ง Force Update อาจจะพอช่วยได้ แต่ไม่แนะนำอยู่ดี

2. How can this workflow be improved? (You can demonstrate any idea, e.g. draw the workflow)

Workflow:



Solve:

- Clone Stage: เปลี่ยนไปควบคุม Pipeline ด้วย Branch main และ Tag เท่านั้น (Source of truth)
    1. ก่อนอื่นบอกก่อนว่า นักพัฒนาจะใช้ Branch dev, sit, uat หรืออื่นๆเหมือนเดิมก็ได้ แต่ Pipeline จะไม่ทำงานในการ Push Branch เหล่านั้น
    2. แบบนี้จะทำให้ไม่มี Branch มากเกินไป ง่ายต่อการควบคุมเพราะทุก Branch จะต้องเข้าไปที่ main เพียงตัวเดียวเสมอ (ไม่ลืมว่า hotfix เข้า uat แล้วแต่ลืมเข้า dev เพราะ hotfix เสร็จก็เข้าไปที่ main เลยอย่างเดียว)
    3. ทาง DevOps บังคับให้ Branch main และ Tag <vx.x.x / x-hotfix> เท่านั้นที่จะทำให้ Pipeline ทำงานได้
    4. การ Push main จะเป็นการ Deploy Environment dev (เท่านั้น)
    5. การ Push Tag (vx.x.x) จะถือเป็นการ Deploy Environment dev -> site -> uat -> production (แบบ Manual ขึ้นอยู่กับความเข้มงวด แต่โดยปกติแล้ว uat + production จะทำเป็น Manual)
    6. การ Push Tag (x-hotfix) จะถือเป็นการ Deploy uat -> production (แบบ Manual โดยข้าม dev -> sit ไป)
- Build Stage: ไม่ใช้ Tag latest ไปใช้ Tag <version / hotfix> / SHA แทน ดังนี้
    1. ใน Stage นี้ Pipeline จะทำการ Build Image ก่อน
    2. ใน Stage นี้จะ Push Tag ขึ้นไปบน ECR ด้วยกันทั้งหมด 3 Tag ได้แก่
        a. Tag: latest
        b. Tag: <version / hotfix> (ถ้าไม่มีจะเป็นชื่อ branch แทน)
        c. Tag: commit SHA
- Deploy Stage: อัพเดท ECS Fargateด้วย Tag ต่างๆ
    3. ถ้า Deploy dev / sit จะใช้ Tag: commit SHA เข้าไปอัพเดทใน ECS Fargate
    4. ถ้า Deploy uat / production จะใช้ Tag: <version / hotfix> เข้าไปอัพเดทใน ECS Fargate
    5. นี่จะทำให้สามารถ Tracking ย้อนดูได้ทุกการ Deploy และย้อน Version ได้ทั้งหมด

6. การ Rollback ก็ไม่ต้อง Run Pipeline ใหม่แค่สลับ Tag เอา

2. The client would like to troubleshoot the slow or unresponsive APIs. What are you supposed to do to advise or demonstrate to the client?

Solve:

1. ตรวจสอบ Health และ Load เบื้องต้น
    a. Resources Usage
        i. CPU / Memory: การใช้ทรัพยากรของระบบ มี Spike หรือไม่
        ii. Load Average: มี Process ที่สูงอย่างต่อเนื่องหรือไม่
        iii. Disk I/: มี Latency สูงเกินไป
        iv. Network I/O: Bandwidth เต็ม หรือ Latency สูง
    b. เครื่องมือ
        i. Prometheus
        ii. Grafana
        iii. AWS CloudWatch
2. ตรวจสอบ Kubernetes Cluster
    a. Pod Health
        i. Horizontal Pod Autoscaler (HPA): มีการตั้งค่าและทำงานได้ปกติหรือไม่
        ii. Pod Restart: บ่อยเกินไปหรือไม่
    b. Node Health
        i. Cluster Autoscaler (CA): มีตั้งค่าและทำงานได้ปกติหรือไม่
    c. เครื่องมือ
        i. Prometheus
        ii. Grafana
3. ตรวจสอบ Behavior API & Tracing
    a. Latency / Throughput
        i. API Latency: สูงผิดปกติ (Response Time = P90+)
        ii. API Throughput: ไหนลดต่ำลงผิดปกติ
        iii. API Error Rate: เพิ่มขึ้นแบบผิดปกติไหม (5xx spike)
        iv. Distributed Service to Service ใช้เวลาเท่าไหร่ ไปช้าที่ Service ไหน
    b. เครื่องมือ
        i. New Relic
        ii. Jaeger
4. ตรวจสอบ Network / Load Balancer
    a. Load Balancer
        i. ALB/ELB: ว่ามี error หรือ timeout หรือไม่
        ii. Gateway: ว่ามี error หรือ timeout หรือไม่
    b. Network Policy

        i. Security Group: มีข้อจำกัด Traffic หรือไม่
        ii. Network ACL: มีข้อจำกัด Traffic หรือไม่
    c. เครื่องมือ
        i. Console
        ii. CloudWatch
        iii. Grafana
5. ตรวจสอบ Logs Monitoring
    a. Logs Monitoring
        i. Error Spike: ถ้ามีตรวจสอบต่อเป็นช่วงเวลาที่เราสงสัยว่ามี Effect API
    b. เครื่องมือ
        i. ELK (หรือ Loki + Promtail + Logstash เบากว่า)
        ii. Grafana
6. (ควรทำ) ตั้ง Alert ล่วงหน้าตามข้อจำกัดต่างๆที่กล่าวมาในข้อ 1-5
7. (Optional) Database Metrics (บางกรณีเพราะส่วนตัวเป็น Backend มาก่อนด้วย)
    a. API Latency: พุ่งสูงเป็นช่วงๆ โดยที่ฝั่ง Infra ยังปกติและโค้ด API ไม่ได้ซับซ้อน อาจเกิดจาก Query Database นี้ช้าเพียงจุดเดียว
    b. Connection Pool: คอขวดทำให้เกิด API Crash ได้ เป็นอีกสาเหตุให้ Pod Restart บ่อย

3. If the client would like to handle 8,500 transactions per second for their services. What are you supposed to ensure their services can rely on to handle it?

Solve:

- **Concept**
    1. Pod Sizing
        - จำนวน Replicas ที่เหมาะสม
        - ผ่านการทดสอบ Load Test ด้วย k6 เพื่อหาว่า Pod แต่ละตัวรองรับกี่ TPS
    2. Nodes & Auto Scaling
        - ใช้ HPA เพื่อรองรับการสเกล Pods
        - ใช้ Cluster Autoscaler เพื่อรองรับการสเกล Nodes
        - อ้างอิงจาก Pod Sizing และมองหา Instance Type ที่รองรับพอดีตั้งแต่ต้น
    3. Message Broker / Queue
        - จัดการ Transaction ที่เข้ามาเป็นจำนวนมากด้วย Queue เช่น Kafka, RabbitMQ
    4. Database
        - Database ต้องมั่นใจว่าเพียงพอต่อการรับ TPS ระดับนี้ได้ด้วย
        - ใช้ Aurora Cluster เพื่อทำ Read Replica
        - Connection Pool ต้องเพียงพอ
        - ใช้ Redis เข้ามาทำ Caching เพื่อช่วยลดภาระของ Database
    5. Observability
        - ติดตั้งระบบติดตาม Latency, TPS, Error ด้วย Prometheus + Grafana / CloudWatch
        - ติดตั้ง Alert ไว้ทุกจุดที่อาจเป็นปัญหา รวมๆคืออ้างอิงคำตอบจากข้อ 2 ได้เลย
    6. Network & Load Balancer
        - Load Balancer (ALB/NLB) รองรับ Throughput / Bandwidth เพียงพอ
        - Instance/Nodes รองรับ Bandwidth เพียงพอ
- **Benchmark Explain**

    Target TPS ต้องมีการเผื่อไว้ โดยต่อจากนี้จะใช้ Target TPS = 10,000 TPS เป็นตัวตั้ง และไม่มีการพิจารณาถึงข้อจำกัดด้านงบประมาณมากนัก

    1. **Benchmark (CPU/Memory)**

        เป้าหมาย
        - ตั้งเป้า P90 Latency
        - Throughput สูงสุดต่อ 1 Pod

        เครื่องมือ
        - k6: Load Testing หา Latency

        ตั้งสมมุติฐาน
        - กำหนดทรัพยากรของ Pod:
            - Max CPU: 500 millicore (0.5 vCPU)

        - Max Memory: 512 MiB
    - กำหนดรูปแบบการยิงโหลด:
        - 1 worker = 1 concurrent user
        - Latency เฉลี่ยต่อ request = 100 ms
        - ยิงพร้อมกัน 50 workers

คำนวณ TPS จากผลทดสอบ

- 1 worker -> 1 req ต่อ 100ms หรือก็คือ 10 req/s (TPS)
- 50 workers -> (50 * 10 TPS) = 500 TPS

คำนวณ Number of Replica

- 10,000 TPS / 500 TPS = 20 Pods
- เผื่อ HA ขึ้นไปอีก 10% -> 20 + 10% = 22 Pods

ผลลัพธ์ (Summary)

- CPU = 22 Replica * 0.5 vCPU = 11 vCPU
- Memory = 22 Replica * 512Mi ~= 12 GiB

ข้อควรระวัง

- สาเหตุที่ไม่เลือกลดจำนวน Replica และอัด Resource เพราะถ้าเกิดเหตุ Pod ล้มแค่ตัวเดียว จะหมายถึงจำนวน TPS ที่ลดลงอย่างมาก
- ค่า Max TPS Per Pod ในความเป็นจริงจะไม่เป๊ะแบบนี้ เพราะ
    - ต่อให้ k6 ทดสอบ AVG Latency <= 100ms
    - แต่ภายใต้สถานการณ์จริง Latency อาจพุ่งสูงถึง 200ms, 300ms
    - แปลว่า หาก latency เพิ่ม เช่น จาก 100 -> 300ms ความสามารถในการรองรับ TPS จะ ลดลงเหลือ ~1/3
    - จะทำให้ CPU และ Memory ที่คาดการณ์ไว้อาจจะไม่เพียงพอต่อให้เผื่อ HA ไว้แล้วก็ตาม
    - ประเมินหน้างาน และงบประมาณว่าสามารถเผื่อขึ้นไปอีกได้ขนาดไหน

2. Benchmark (Network I/O)

เครื่องมือ

- k6: Load Testing เพื่อเอา data_sent / data_received / http_reqs
- Prometheus + Grafana: ถ้าต้องการเอาข้อมูลนี้ไปทำ Dashboard

ตรวจสอบหาขนาด Data Per Transaction

- แต่ละ API endpoint มี request/response size ไม่เท่ากัน
- เลือก API ที่ ใช้บ่อยที่สุด และ ใหญ่ที่สุด
- สุ่มเลือก API ขนาดกลางอื่นๆ
- คำนวณขนาด request + response (req/res) เฉลี่ยต่อ transaction

ตั้งสมมุติฐาน

- req/res รวมกัน ~= 1 KB

คำนวณ Bandwidth ที่ต้องใช้

- เอา Target TPS * (req/res)
- จะได้ 10,000 * 1 KB = 10 MB/s

ตรวจสอบ Infra ว่ารองรับเพียงพอหรือไม่

- Load Balancer (ALB/NLB)
  - ตรวจดู Bandwidth Quota เพียงพอหรือไม่
  - ปกติ AWS รองรับได้อยู่แล้วถ้าแค่ 10MB/s แต่ถ้าในความเป็นจริงมากกว่านี้ต้องไปตรวจสอบดูอีกที
- Envoy Proxy (ถ้าใช้ sidecar:
  - max_response_bytes ต้อง >= 1 KB
  - max_request_bytes ต้อง >= 1 KB
  - max_connections ต้องเพียงพอต่อ 10 MB/s (คำนวนดูได้โดยการใช้ k6 หา throughput ((req+res) / duration) แต่ลองปรับแล้วยิง k6 จริงเลยไวกว่า)

- **Node Selection & Scaling Strategy**

1. **คำนวณ Resource Requirement**

จาก Pod Benchmark

- CPU: 22 Pods × 500m = 11 vCPU
- Memory: 22 Pods × 512 MiB ≈ 12 GiB

เผื่อ Overhead 20% (OS + kubelet + system pod)

- CPU: 11 + 20% ~= 14 Core
- Memory: 12 + 20% ~= 15 GiB

Bandwidth

- 10 MB/s

Number of Pods

- 22 pods

2. **เลือก Node Type**

AWS EC2 (c6a.4xlarge)

- vCPU: 16 Core
- Memory: 32 GiB
- Max Pods: 58 pods (ENI limit)
- Bandwidth: 10 Gbps (~=1,250 MB/s)

High Availability (HA)

- จำนวนแนะนำ 2 Nodes
  - vCPU รวม: 16 * 2 = 32 Core
  - Memory รวม: 32 * 2 = 64 GiB

        - Max Pods รวม: 58 * 2 = 116 pods
        - Bandwidth รวม: 10 * 2 = 20 Gbps
- เหตุผลที่เลือกใช้ Type ที่เผื่อ Resource ที่ดูเกินความจำเป็น
    - เพิ่ม Fault Tolerance (ถ้า Node ล้มไป 1 ตัว ยังมีอีกตัวรับโหลดได้)
    - การบำรุงรักษาง่ายกว่า ลดปัญหาจุกจิก
    - ยิ่ง Node มีขนาดใหญ่ Overhead Impact จะน้อยลงอีก ทำให้ในความจริงแล้วเรามี Resource เหลือมากกว่าที่คำนวณนี้อีก

4. Write a Terraform script using the diagram below. (Please do a Zip file and attach with the answer document file.)

Solve: ก่อนอื่นจาก Diagram ที่ให้มา ตามที่เข้าใจ `private_subnet_cidrs` ใช้ CIDR Block ที่อยู่นอก VPC CIDR (`172.25.0.0/16`) ซึ่งจะทำให้ Subnet Provisioning ล้มเหลวใน AWS เพราะ subnet ควรอยู่ภายใต้ VPC CIDR เดียวกัน เลยไม่แน่ใจแต่เบื้องต้นผมเลยเปลี่ยน `private_subnet_cidrs เป็นตามนี้ก่อนครับ`

```
[private_subnet_cidrs]
"ap-southeast-1b" = "172.25.1.0/24"
"ap-southeast-1c" = "172.25.2.0/24"
```