Containerise an Existing Application

By Tyson Williams

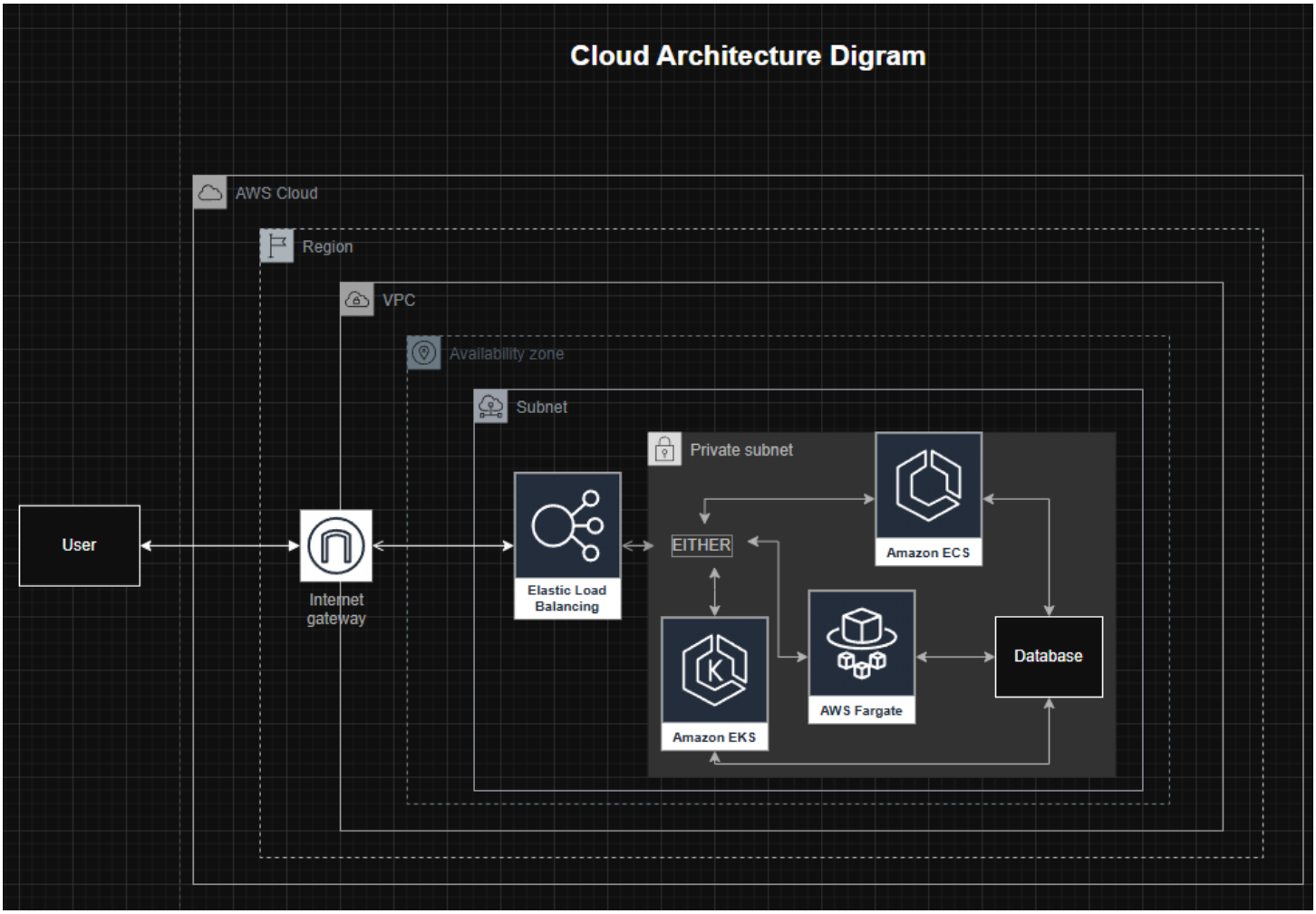

Fig.1 Draw.io diagram of the cloud architecture diagram

Cloud Architecture Diagram Explanation (AWS, Containerised API)

Component Breakdown and Interaction

1. User
    - Who/What it is: The end-user making HTTP requests to the application
    - Interaction: Sends requests from a client (browser, Postman, Bruno, etc.)

- Outcome: These requests are routed through the internet to AWS

2. AWS Cloud
    - What is it: The overarching infrastructure provided by Amazon Web Services
    - Interaction: Hosts all resources and services required for deploying and running the application
    - Outcome: Serves as the platform where all resources are created and managed

3. Region
    - What is it: A geographically isolated AWS location (e.g ap-southeast-2 for Sydney)
    - Interaction: You select the region closest to your users to reduce latency
    - Outcome: All AWS services (like VPCs, EKS, EC2, etc.) are launched within a selected region

4. VPC (Virtual Private Cloud)
    - What is is: A logically isolated network within the AWS Cloud where you define and control your infrastructure
    - Components:
        - CIDR block: IP range for your resources
        - Route Tables: Determine traffic flow
        - Security Groups and NACLs: Handle network-level security
    - Interaction: All your AWS resources (EKS, ECS, Fargate, DBs) are deployed inside the VPC
    - Internet Gateway: Enables communication between instances in the VPC and the internet (eg. for public-facing APIs)
    - Outcome: Provides networking, routing and firewall capabilities for secure communication

5. Availability Zone (AZ)
    - What it is: A physically separate data center within a region
    - Interaction: Used for high availability, you can deploy your services across multiple AZs to ensure redundancy and fault tolerance
    - Outcome: Helps avoid single points of failures in case of data center issues

6. Subnet
    - What is it: A segment of the VPC where resources (containers, load balancers, databases) are placed
    - Types:
        - Public Subnet: Accessible from the internet (e.g. load balancer)
        - Private Subnet: Not internet-facing (e.g. API containers, databases)
    - Interaction: Subnets are assigned to AZs and define how resources are exposed or secured
    - Outcome: Organises and controls traffic flow to specific resources in the VPC

7. Elastic Load Balancing (ELB)
    - What it is: A managed AWS service that automatically distributes incoming traffic across multiple targets (e.g. containers or EC2 instances)
    - Types: Application Load Balancer (ALB), Network Load Balancer (NLB), Gateway Loan Balancer (GWLB)
    - Interaction:
        - Receives inbound traffic from users via the internet
        - Routes HTTP(S) requests to the containerised backend running on EKS/ECS/Fargate
        - Performs health checks to ensure traffic only goes to healthy services
    - Outcome: Improves scalability, fault tolerance, and provides a single entry point to the application

8. Amazon EKS / ECS / AWS Fargate
    - What why are:
        - Amazon EKS: Managed Kubernetes service, best if you're using Kubernetes for container orchestration
        - Amazon ECS: AWS-native container orchestration, simpler than EKS
        - AWS Fargate: Serverless container compute engine, no need to manage EC2 instances
    - Interaction:
        - Hosts and runs your containerised API applications
        - Pulls container images from Amazon ECR (or Docker Hub)
        - Scales up or down based on traffic
        - Works with the Load Balancer to receive requests and return responses
    - Outcome: Provides the compute layer that runs your backend application logic in isolated, scalable containers

Summary of Interactions:

1. User sends a request, reaches AWS via internet
2. AWS Region routes it to your VPC
3. Traffic flows through Internet Gateway into a public subnet
4. Elastic Load Balancer receives the request and forwards it to an API container inside EKS/ECS/Fargate in a private subnet
5. The container processes the request, may query a database, and returns a response through the same path

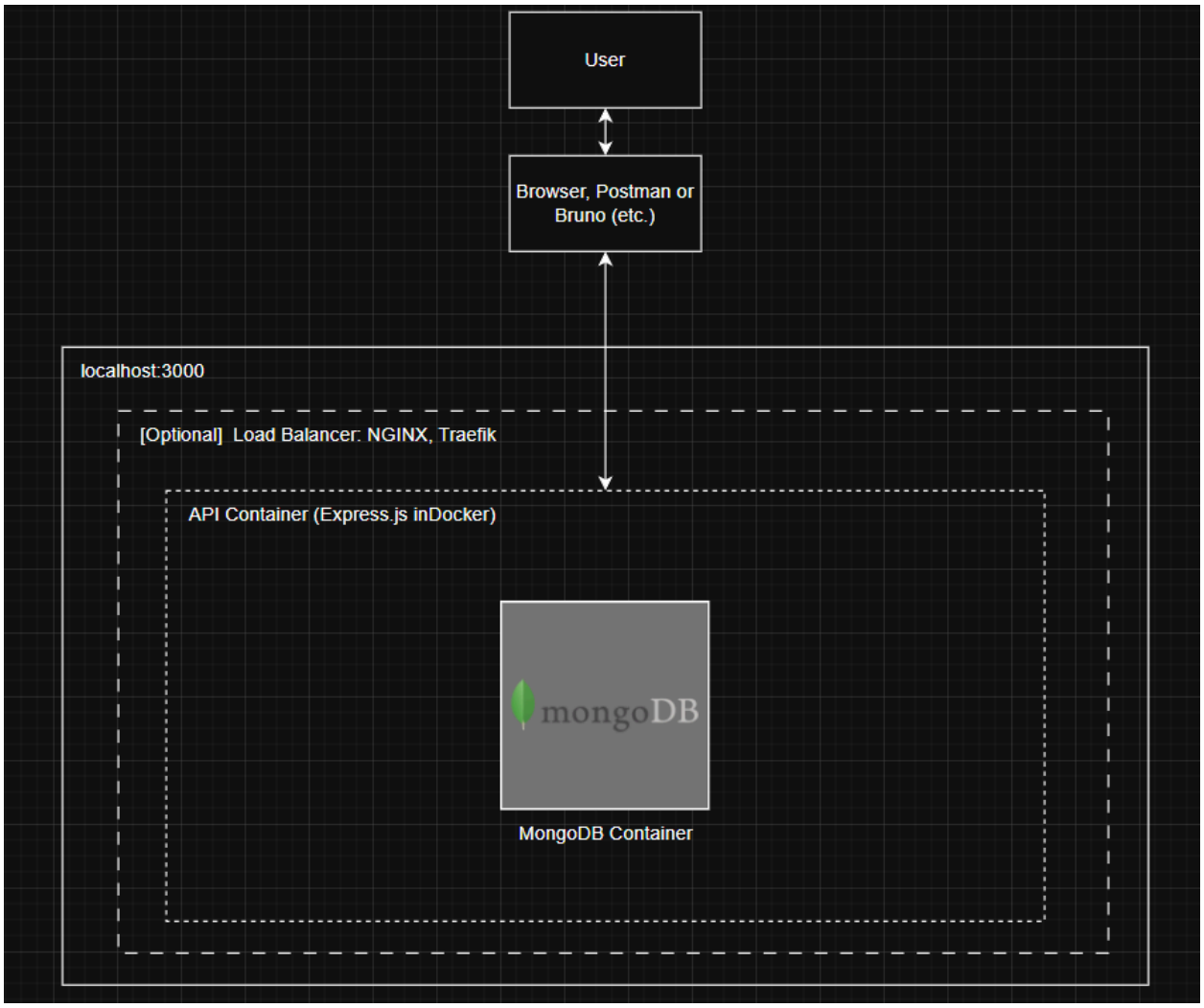

Fig. 2 Local Container Architecture

Local Application Architecture Explanation

Component Breakdown and Interaction

1. User
    - What it is: A developer or tester making HTTP requests to the application
    - Interaction: Sends requests via tools like Bruno, Postman, or a web browser
    - Outcome: Initiates interaction with the backend API, triggering the container stack to respond

2. Bruno / Postman / Browser
    - What it is: client-side applications used to test or interact with APIs

- Interactions: Sends HTTP requests (GET, POST, etc) to localhost:3000
- Outcome: Interfaces directly with your application as if it were in production, used for development, debugging, or automated testing

3. Localhost:3000 (Host Machine's Network Interface)
    - What is it: The local network interface that maps a port from a Docker container to the host machine
    - Interaction:
        - Requests made to localhost:3000 are forwarded to the API container via Docker port binding (e.g. ports: ["3000:3000"] in docker-compose.yml)
        - Enables tools outside the Docker network to access services
    - Outcomes: Acts as a bridge between the user's tools and the containerised API

4. Local Load Balancer (Optional - NGINX, Traefik)
    - What it is: A reverse proxy that can forward requests and routes them to the correct container (useful for simulating production environments)
    - Interaction:
        - (If included) Handles incoming requests and routes them to the correct container (useful for simulating production environments)
        - May provide HTTPS termination, routing, and basic security
    - Outcome: Optional component that adds flexibility, simulates real-world load balancer behaviour

5. API Container (Docker)
    - What it is: A Docker container running your backend service, usually built with Express, Fastify, Flask, etc.
    - Interaction:
        - Listens on a port (e.g. 3000) inside the Docker network
        - Handles requests coming from localhost or the test container
        - Queries MongoDB, processes data, and returns responses
    - Outcome: The core logic layer of your application, it receives, processes, and responds to user requests

6. MongoDB Container (or local MongoDB instance)
    - What it is: A database service running in a separate Docker container (or natively on your machine)
    - Interaction:
        - The API container connects using a connection string like mongodb://mongo:27017
        - Receives read/write operations from the API
        - Stores structured data such as user info, posts, product info, etc.
    - Outcome: Stores and retrieves application data during runtime, testing, or development

Summary of Interactions

1. The User initiates a request through Postman/Bruno/Browser to localhost:3000
2. Localhost:3000 forwards the request into the Docker network via port binding
3. The API container receives the request and processes it
4. If needed, the API communicates with the MongoDB container to perform data operations
5. The response is sent back through the stack to the User

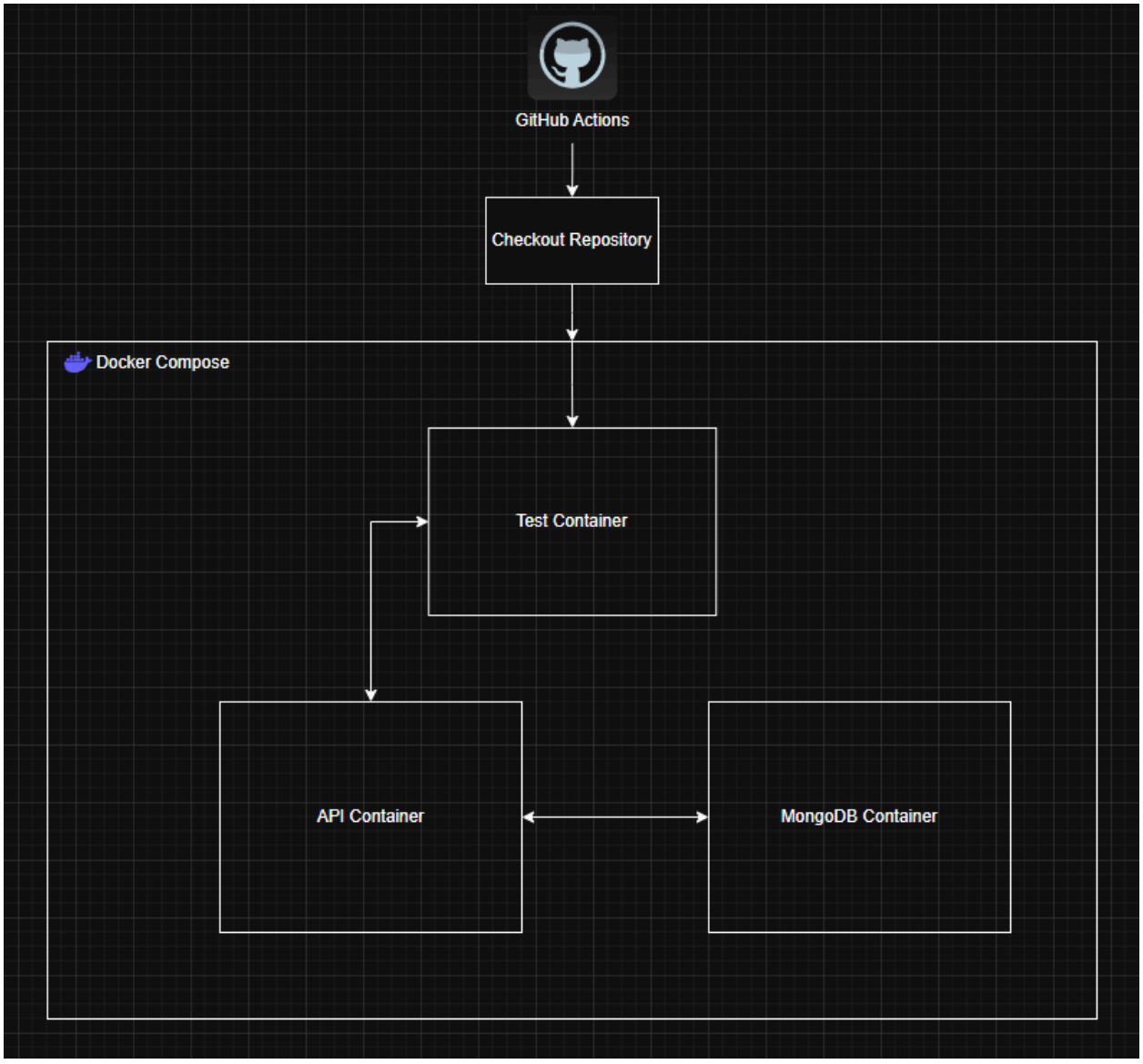

Fig. 3 Testing Architecture Diagram through GitHub Actions

Testing Architecture Diagram - Docker Compose

Component Breakdown and Interactions

1. GitHub Actions
    - What it is: A cloud-based CI/CD automation platform provided by GitHub
    - Interaction:
        - Spins up a temporary Ubuntu runner to execute your CI workflow (e.g. github/workflows/test.yml)
    - Outcome: Automates testing of your application every time you push code, ensuring reliability and preventing regressions

2. Checkout Repository
    - What it is: A step in the GitHub Actions workflow that pulls your source code into the GitHub runner
    - Interaction:
        - Uses actions/checkout@v4 to copy your app's repo to the CI environment
    - Outcome: Makes all project files (e.g. Dockerfiles, docker-compose.yml, tests, source code) available for container build and execution

3. Docker Compose
    - What it is: A tool for defining and running multi-container Docker applications
    - Interaction:
        - Uses your docker-compose.yml file to spin up the test environment, including:
            - API container
            - MongoDB container
            - Testing container
    - Outcome: Creates a self-contained, reproducible test environment inside the GitHub Actions runner

4. Test Container
    - What it is: A Docker container built specifically to run your test suite (e.g. using Jest, Mocha, Supertest, etc.)
    - Interaction:
        - Starts after the API and DB are ready
        - Sends HTTP requests to the API container
        - Performs assertions on expected behaviour (status codes, DB state, response content)
    - Outcome: Verifies that your application behaves as expected, catching bugs before deployment

5. API Container
    - What it is: The same backend container used in development (e.g. Express.js running in Node)
    - Interaction:
        - Listens for incoming HTTP requests on a port
        - Communicates with the MongoDB container for read/write operations
        - Returns responses back to the Test container
    - Outcome: Serves as the backend logic layer under test

6. MongoDB Container
    - What it is: A container running a MongoDB instance used only for testing purposes
    - Interaction:
        - Accepts requests from the API container
        - Provides a clean, isolated database for each test run
    - Outcome: Mimics the real database environment without polluting production or dev data. It's wiped clean between CI runs

Summary of Interactions:

1. GitHub Actions launches a runner and checks out the code
2. Docker Compose builds and starts the containers (API, MongoDB, Test)
3. The test container:
    - Waits for the API to be ready
    - Sends test requests to the API container
    - API communicates with MongoDB
    - Results are returned and verified
4. When complete, the runner tears everything down, ephemeral and isolated

References

Amazon Web Services (AWS), 2024. *What is AWS?* [online] Amazon Web Services, Inc. Available at: https://aws.amazon.com/what-is-aws/ [Accessed 15 July 2025].

Amazon Web Services (AWS), 2024. *Regions and Availability Zones*. [online] Amazon Web Services, Inc. Available at: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html [Accessed 15 July 2025].

Amazon Web Services (AWS), 2024. *Amazon VPC Documentation*. [online] Available at: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html [Accessed 15 July 2025].

Amazon Web Services (AWS), 2024. *Elastic Load Balancing*. [online] Available at: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html [Accessed 15 July 2025].

Amazon Web Services (AWS), 2024. *Amazon ECS, EKS, and AWS Fargate Overview*. [online] Available at:

- ECS: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html
- EKS: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
- Fargate: https://docs.aws.amazon.com/fargate/latest/userguide/what-is-fargate.html [Accessed 15 July 2025].

Docker, 2024. *Docker Overview*. [online] Docker Inc. Available at: https://docs.docker.com/get-started/overview/ [Accessed 15 July 2025].

Docker, 2024. *Docker Compose Documentation*. [online] Docker Inc. Available at: https://docs.docker.com/compose/ [Accessed 15 July 2025].

MongoDB, 2024. *MongoDB Docker Official Image*. [online] Available at: https://hub.docker.com/_/mongo [Accessed 15 July 2025].

GitHub, 2024. *About GitHub Actions*. [online] GitHub, Inc. Available at: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions [Accessed 15 July 2025].

GitHub, 2024. *Running Docker Containers in GitHub Actions*. [online] GitHub, Inc. Available at: https://docs.github.com/en/actions/using-containerized-services/about-service-containers [Accessed 15 July 2025].