Overview

From the problem description and my prior email correspondence, I believe you're looking for the following:

- A basic implementation of the Guestbook App
  - Must deployed via Vagrant, but provisioned however I see fit.
  - Include Vagrantfile and any other provisioning files/scripts/recipes
  - This simple example need not have all features to be highly available, scalable, etc…
- A diagram and description of how I'd architect a highly available, scalable, and maintainable version of the application.

I intend to use Docker as the provisioning system and will deploy to AWS.

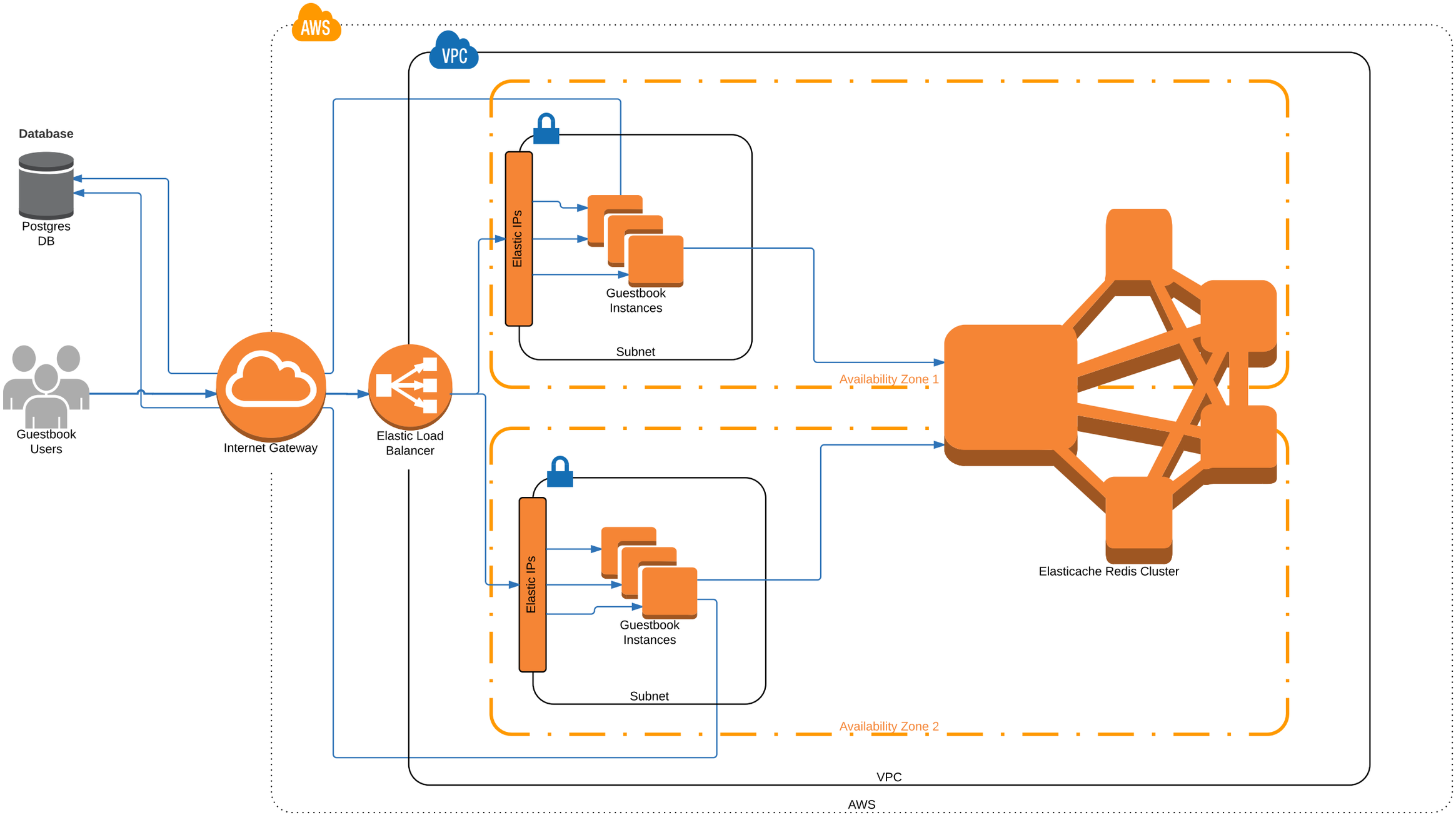
Assumptions

- HTTPs is unneeded for this application as data a user inputs is readily available on the welcome page immediately after submission. If we decide to add features that may depend on sensitive information, in the final deploy we'd implement HTTPS termination at the Elastic Load Balancer (ELB)
- There is AWS account is already created and we have access to the following:
  - AWS access key credentials with sufficient permissions to provision & modify existing resources
  - A public/private key pair to use for SSH access to EC2 instances.
    - This key pair should be uploaded/available in the region you're planning to deploy this application.
    - Whatever environment is orchestrating the provisioning—your local machine or Jenkins CI for example—has access to the private key file.
- Inside the AWS account, there is an established VPC to deploy into with:
  - An Internet GW setup
  - Subnets established
  - Routes setup correctly
  - A default security group for VPC
    - Allow systems inside the VPC to establish outgoing connections (to the Postgres DB for example)
    - All inter VPC traffic is allowed between members of this security group for simplicity of this exercise
    - Will allow incoming port 80 requests (only the guestbook instances and/or their ELB will be listening)
- The Postgres DB is accessible from the VPC
  - SSL enforcement iscontrolled by postgres preference, Ruby should negotiate as needed

    - The Postgres system is high availability (assume it won't be the weakest link in system)
- Need to assume cannot modify the Ruby application necessary for deployment needs now or in future!
- The Ruby Guestbook Application only allows providing a single database URL and a single redis host url.
    - High-availiability must occur at the respective service
- The instances have access and can pull Docker containers from Docker hub or whatever image repo is required for provisioning
- The orchestration system has docker and vagrant already installed and functioning.

Minimal Version Deploy Instructions

- Install/Use Vagrant AWS provider module and install a dummy box:
    - `vagrant plugin install vagrant-aws`
    - `vagrant box add dummy https://github.com/mitchellh/vagrant-aws/raw/master/dummy.box`
- Use Docker to build the Dockerfile provided if you don't want to use the already built version I've put on Docker hub
    - To build:
        - `docker build -f /path/to/Dockerfile -t <repo>:<tag> /path/to/guestbook`
- Edit the Vagrantfile and add data for the following in the provider block:
    - `aws.access_key_id = "<ACCESS KEY ID>"`
    - `aws.secret_access_key = "<SECRET ACCESS KEY>"`
    - `aws.keypair_name = "<KEYPAIR FOR SSH TO INSTANCE>"`
    - `aws.region = "<AWS REGION>"`
    - `aws.subnet_id = "<SUBNET ID>"`
    - `aws.security_groups = ['<VPC SECURITY GROUP>']`
    - `override.ssh.private_key_path = "<LOCAL PATH TO PRIVATE KEY>"`
- In the Docker provisioner block change the docker run args to have a valid DATABASE_URL
- After this all set, just type:
    - `vagrant up --provider=aws`


AWS
VPC
Database
Postgres DB
Guestbook Users
Internet Gateway
Elastic Load Balancer
Elastic IPs
Guestbook Instances
Subnet
Availability Zone 1
Elastic IPs
Guestbook Instances
Subnet
Availability Zone 2
Elasticache Redis Cluster
VPC
AWS

Design Choices

- Scalability
  - The only limitations are internal bandwidth concerns in AWS and scalability of Postgres
  - The EC2 guestbook application servers would be in autoscaling groups and could be set to increase/decrease as needed.
    - Would need to add additional step of creating an AMI for the final provisioned instance, this would be set for the launch config used by the autoscale (no need to go through the provisioning multiple times!)
  - In full deploy, using Elasticache in cluster mode would allow for scale horizontally as needed with Redis
    - Reason for cluster—all instances need access to shared key/value. Slight overhead here vs. redis on the instance (like my minimal deploy) but this won't a major issue.
    - Doubt need for many instances due to the limit of data stored (one key with single integer value)
    - Would need a large enough node instance to allow for Moderate network traffic to support traffic demand of scaling EC2 Nodes eventually
- Performance
  - The architecture as defined give adequately sized AWS instances (they tie network performance to instance size) should do well
    - As demand grows, scaling horizontally the EC2 nodes and Redis cluster nodes should allow for maintaining performance
  - As noted in the problem case, latency for Postgres is a non-concern, however, in reality, this would be a performance bottleneck given this architecture.
- Realiability
  - All components are redundant
    - Multi-zone ELBs
    - Only 2 Zones defined in diagram, but we could add more
    - Multiple EC2 Guestbook instances
    - Multiple Elasticache (redis) nodes in cluster mode (auto-failover & load balancing)
  - Once again Postgres appears to be primary concern
- Security
  - I would classify this as moderately high protection for the needs of the applicatiion
    - Outside SSH being open for the orchestration system, only port 80 available to public to hit ELB (or instance in minimal deploy)
    - Might leave port 80 blocked outside VPC for instances, create 2nd security group for ELB with just port 80. Then give access to port 80 in VPC group to only members of the ELB's new security group

    - Only security concern is once inside VPC, you're home-free port-wise, but only vectors are in is through an exploit on port 80 of the EC2 Instance, through the ELB, or in via the provisioning IP on port 22
        - Both non-trivial to exploit if reasonable precaution taken to protect
        - Could add additional monitoring of traffic for malicious attacks
    - Realistically, it's all a balance of what the value is vs. the cost/time/effort to harden. This architecture is sufficient for the current guestbook app.
- Maintainability
    - Using docker means, we don't need to worry much in terms of maintaining the underlying Ubuntu boxes.
        - We've installed nothing but Docker on them, each time we provision, we get the latest stable Docker.
        - Just need to update base AMI as we go to get security patches, etc… a well vetted/maintained base AMI (like amazon linux or ubuntu official) let's
- Cost
    - Want cheap, we can lose the multizone and redundancy, but it's a tradeoff for reliability
    - The primary cost concern I see is with the Elasticache cluster, these can get $$$$ fast. However, if we need reliability we need a cluster of nodes. Manually setting this up on EC2, you're still paying for the instances, plus you're eating development cost to build, maintain, etc...
- BONUS: Portability
    - The use of docker allows easy leaving of AWS as long as we've a Vagrant provider for whichever other service we want to use
    - Same system/setup can be run on developers local machine