SRE

[Assumptions](#_ssgu5gvncsey)

[High level design](#_8ihbbep9ghuh)

[Infrastructure as Code (IaC)](#_lsgvlqeroopc)

[Infrastructure/Server Automation](#_v09ts9vblhwd)

[Infrastructure definition tool](#_yek2mdvdw9xd)

[Server configuration definition tool](#_7tz88srf3npw)

[Server templating and packing tool](#_zcrbm38siay1)

[Server validation tool](#_4bo1p9w2db98)

[Server change management Mode](#_gbxssot90tej)

[Local testing](#_w46koxdurhpx)

[CI pipeline](#_hoq5s9q6a766)

[CD pipeline](#_ox0g0n27qdep)

[Centralized logging](#_b8gha4275o00)

[Overview (efforts: 5-10 man-day)](#_y25lw7tm9484)

[Scaling (efforts: dependence on approach, 5-10 man-day)](#_hk009e9toci4)

[Security (efforts: 2-3 man-day)](#_j7gy1ji76og5)

[data encryption at transit](#_v4xycaf9b3b)

[Data encryption at rest](#_j0fitnkgu1w9)

[Authentication and Authorization](#_9ggw2u8ny7bd)

[Monitoring and altering](#_yli8zh564orv)

[Overview (efforts: 4-5 man-day)](#_6ye3oxmnxhwy)

[Scaling (efforts: 1 man-day. Since using IaC, just need to apply same code to a different server)](#_y1xundt0h65j)

[Security ( efforts: 2 - 3 man-day)](#_ktw389k6uh9p)

[Data encryption at transit](#_rrnsuv90xi2p)

[Data encryption at rest](#_i577g4lusahp)

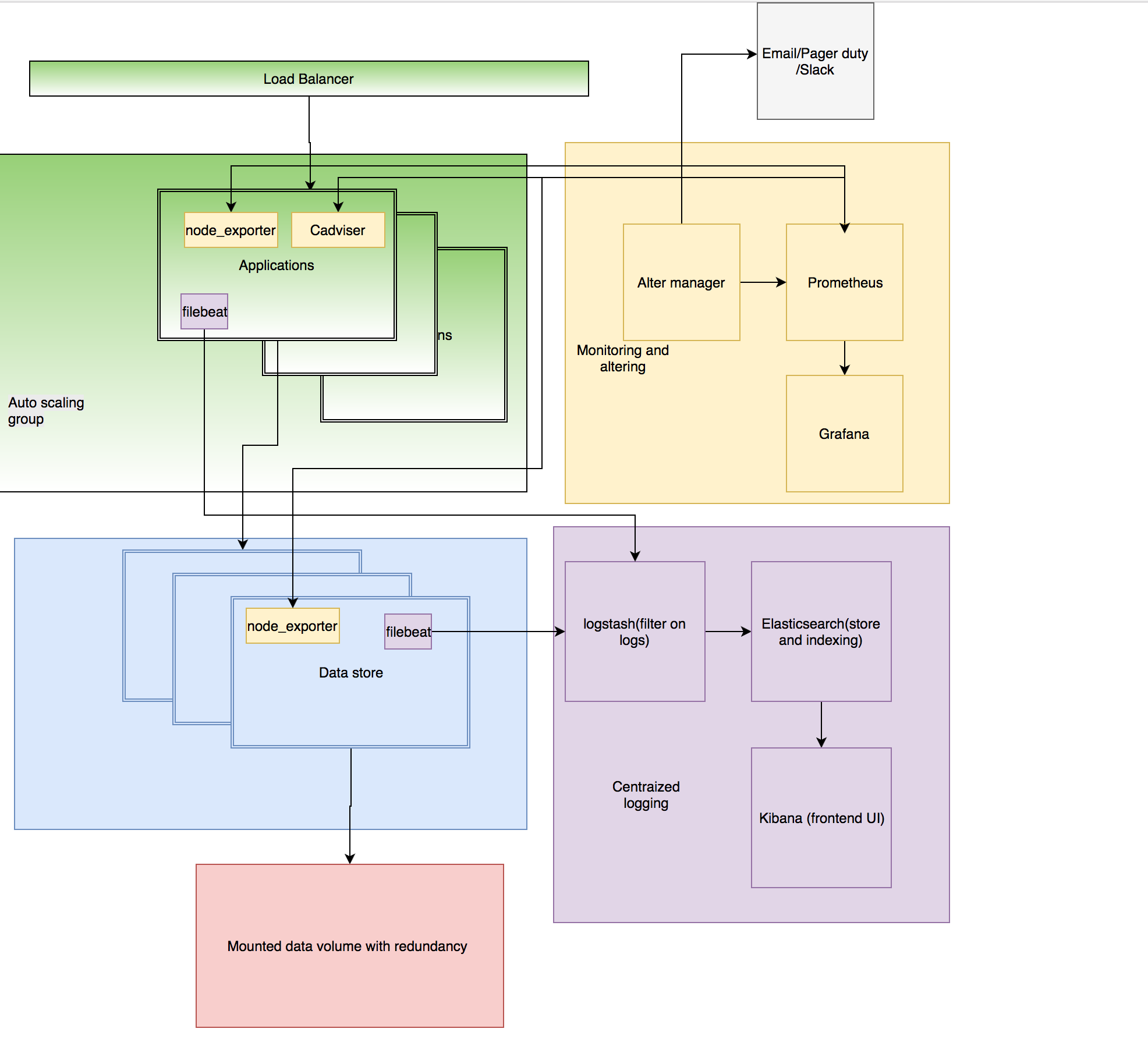
[Authentication and Authorization:](#_t3wt6rm166fi)

Solution

# Assumptions

1. The environment will be a (public) cloud environment. The reason why (public) cloud is picked over private cloud is due to several benefits
   1. Elasticity -- allow apps to quickly scale up to maintain SLA and scale down to save the cost
   2. Disposable -- Pet vs Cattle analogy. Instead of given special care to vms that are malfunctioning. Destroy and provision a new vm could save time.
   3. Availability -- Public cloud provides zone/datacenter availability in a cost effective manner compared to private cloud
   4. Management overhead -- Public cloud provide the base infrastructure ( Networking/ disk volume/ Hypervisor) as a service which reduces management overhead.
2. The workload is spread to application server using AWS ELB -- application load balancer.
3. Application is running is docker container
4. Centralized logging used is ELK stack, Elasticsearch, logstash and Kibana
5. Centralized logging used is Prometheus

# High level design



The way to reduce the time spend on this task using automation. To be specific, using infrastructure as code approach to automate the environment provision. This will also allow us to build the entire environment from scratch.

Extra load handling is defined in the scale section of each solution

# Infrastructure as Code (IaC)

IaC enables consistent, repeatable routines for provisioning and changing systems and their configuration.

Changes that made to definitions and then rolled out to systems through unattended processes that include thorough validation.

All the changes make to the infrastructure is peer reviewed and tracked in version control system which is available for roll-back and auditing.

## Infrastructure/Server Automation

### Infrastructure definition tool

Terraform is used to define and automate cloud infrastructures.

Code can be found [here](https://github.com/SiweiWang/SRE/tree/master/terraform)

In this code, application is in auto scaling group which will scaling in and out using scaling policy. Current scaling policy is to scale out during business hour and to scale in during non-business hour.

Code is also using terraform module to define different environment. For example, Dev environment will not auto scale but production will auto scaling from 1 to 10 instance.

There is also load balancer in front of the auto scaling group which will balance the load for the auto scaling group. It also do health check and avoid traffic to get sent to unhealthy nodes.

### Server configuration definition tool

Ansible is used as the server configuration definition tool.

Code can be found [here](https://github.com/SiweiWang/SRE/tree/master/ansible)

There are different roles defined in the code here. Commons are the stepping stone for the applications. Apps dir container application related roles. App and data are place holder for application and data store code with monitoring and logging agent installed. Monitoring and logging are the two role that is defined. Both are using docker compose to spin up the whole stack for monitoring server and logging server.

### Server templating and packing tool

Packer

Packer can be use to pack the server into a VHD and then uploaded as a image to use.

**Backlog**: create server template in packer to speed up the provision process and immutable infrastructure

### Server validation tool

Server Spec is used as server validation tool that allows

**Backlog:** implement server validation using server spec

### Server change management Mode

We are using immutable Infrastructure to handle updates of infrastructure and application

* New Infrastructure every deployment
* Blue green deployment deployment

### Local testing

Local testing of configuration management is vagrant code is [here](https://github.com/SiweiWang/SRE/tree/master/vagrant)

Example Workflow:

1.Check out the latest versions of the Vagrantfile, Ansible playbooks, and test scripts from VCS.

2. Build the local VMs (vagrant up).

3. Run the automated test scripts to make sure everything works.

4. Write an automated test for the change you plan to make.

5. Edit the Ansible playbooks to implement the change.

6. Apply the change to the local VMs (vagrant provision).

7. Run the tests to see if they work. Repeat steps 4 through 7 until happy.

8. Update the files from VCS again, to merge any changes that other people have made since you checked out in step 1.

9. Destroy and rebuild the local VMs (vagrant destroy -f ; vagrant up) and run the tests again to make sure everything still works after the merge.

10. Commit your changes to VCS.

11. Verify that the changes pass successfully through the pipeline’s automated stages.

12. Once the changes reach production, verify that everything works the way you wanted.

# CI pipeline

**Backlog**: CI pipeline for application

# CD pipeline

**Backlog**: CD pipeline for application

# Centralized logging

## Overview (efforts: 5-10 man-day)

Centralized logging is achieved using ELK and centralized logging server. It is open source software which allows logging filtering (logstash) and index( elasticsearch). It also has a nice UI layer that build on top for easy human interaction. ELK stack is designed with scaling in mind each layer can scale independently. Both logstash and elasticsearch can scale as a cluster and have multiple node in the cluster. The Kibana is stateless UI frontend which can also scale out easily.

Each node will have filebeat installed to ship required logs.

## Scaling (efforts: dependence on approach, 5-10 man-day)

Both logstash and elasticsearch are resource intensive and can become bottleneck which slow down the log processing. As the back pressure building up, it could crash logstash and/or elasticsearch by the sheer volume of the input. One way to help with back pressure is to use a queue to temporarily hold the logs as a mechanism to avoid overload logstash.

Another way way to scale logging is following lambda architecture for data streaming. All the log is stream to cloud storage (for example AWS s3) before processing. And then the logs is shipped from S3 to the ELK cluster for processing to avoid overload the cluster. Once log is processed, it can be move to aws glacier for cost optimization. The advantage of this architecture is it reduce the risk of losing data since S3 is redundant. ELK stack can scale dynamically depends on the volume of the log. The drawback is that there will be a delay on the log analysis.

## Security (efforts: 2-3 man-day)

### data encryption at transit

TLS is supported to encrypt the data at transit

### Data encryption at rest

Supported using disk encryption

### Authentication and Authorization

Elastic does offer security plugin for file based/LDAP authentication and authorization. It will secure elasticsearch and Kibana.

IP tables can also used for locking down access to specific ip range.

# Monitoring and altering

## Overview (efforts: 4-5 man-day)

Prometheus is used as monitoring and alerting server. When it comes to monitoring, there are couple of level to consider

1. vm metrics/monitoring.
   1. Node exporter is used to get vm metrics( such as disk IO/ Memory/ CPU utilization)
2. Container monitoring
   1. cAdviser is used to get container metrics
3. Application monitoring
   1. application monitoring /performance monitoring (APM). This is to get metrics related to the application. This require application to expose instrumentation/metrics. For example, JMX metrics is used to monitor JVM metrics

Prometheus has couple of advantages here

* Part of cloud native platform
* Works with multiple backend ( consul, etcd, docker swarm, k8s)
* Native support grafana and cAdviser
* Time series database allows query in stas, useful for creating alerts
* Comes with alerting manager to send alerts to different source
* Huge selection of open source metrics exporter for different applications

## Scaling (efforts: 1 man-day. Since using IaC, just need to apply same code to a different server)

Prometheus does not support clustering. Right now only can scale up however, it can not scale out.

If one box can not handle the load, multiple instance of Prometheus might required to support different part of the platform.

## Security ( efforts: 2 - 3 man-day)

### Data encryption at transit

TLS should be used to encrypt the data at transit

### Data encryption at rest

Supported using disk encryption

### Authentication and Authorization:

Grafana supports based auth.