Final Assignment

Detailed Design - Payment Processing System

for

Master of Science

Information and Communications Technology

Distributed Computing

ICT 4310-1

Richard Magill

University of Denver University College

May 22, 2022

Faculty: Mike Prasad

Director: Michael Batty, PhD

Dean: Michael J. McGuire, MLS

Contents

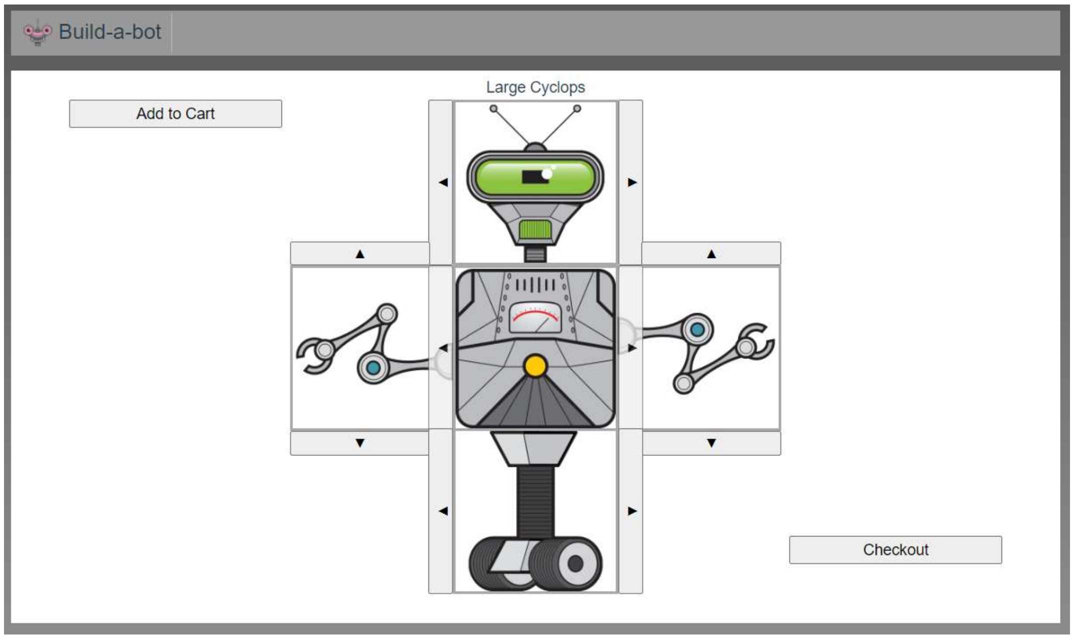
Introduction

This paper describes the design for a distributed eCommerce web application integrated with a credit card payment processing system. The eCommerce application itself is a Vue.js web app called Build-a-Bot that allows site users to select robot parts and then order a completed robot. The Vue.js client sends the order (in the form of an HTTP Post) to a Node.js proxy server, which then formats it and redirects the request to a Stripe.com checkout page. A prototype of the design was successfully built and is now ready for deployment to a Test environment.

Two designs to host the application were explored as part of this analysis, one using Azure cloud components and an Infrastructure-as-a-Service (IAAS) approach, and the second using Heroku Dyno containers as Platform-as-a-Service (PAAS). While either concept is technically feasible, for many reasons (including cost, scalability, and fault tolerance), my recommendation is to go with the containerized PAAS design. In the following pages I will describe the application, the proposed IAAS and PAAS designs, and will explain my rationale for recommending the PAAS approach with Heroku.

eCommerce Website (Vue.js)

The website is a Vue.js application called Build-a-bot. It is based primarily on a Pluralsight course, "*Vue.js Fundamentals*" by Jim Cooper (Pluralsight.com, ND). The site's home page allows a user to select various robot parts and build a robot of their choice. The Pluralsight course covered the "robot" elements, while I subsequently added the checkout flow via Node.js and Stripe.com. The following screenshot shows the application with an example robot type selected:



Once the user has selected all the desired parts, the “Add to Cart” button will place the selected robot into a shopping cart. The user can then “Checkout”. The app utilizes an HTML Form POST method with a body that provides the details of the shopping cart to the Node server (note I have implemented environment variables so the url is not hard-coded).

```
        <form :action= nodeUrl.url method="POST">
          <div class="cost">
            <div v-for="robot, index in cart" :key="index">
              <input type="text" id="price-input" v-model= robot.price
name="price" hidden/>
              <input type="number" id="quantity-input" min="1"
                value="1" name="quantity" hidden />
            </div>
          </div>
```

HTTP Proxy Server (Node.js)

The Node server runs in the background, and is responsible to receive the Post from checkout and then formats the request body for Stripe (masteringjs.io, 2022). It then redirects the request to the

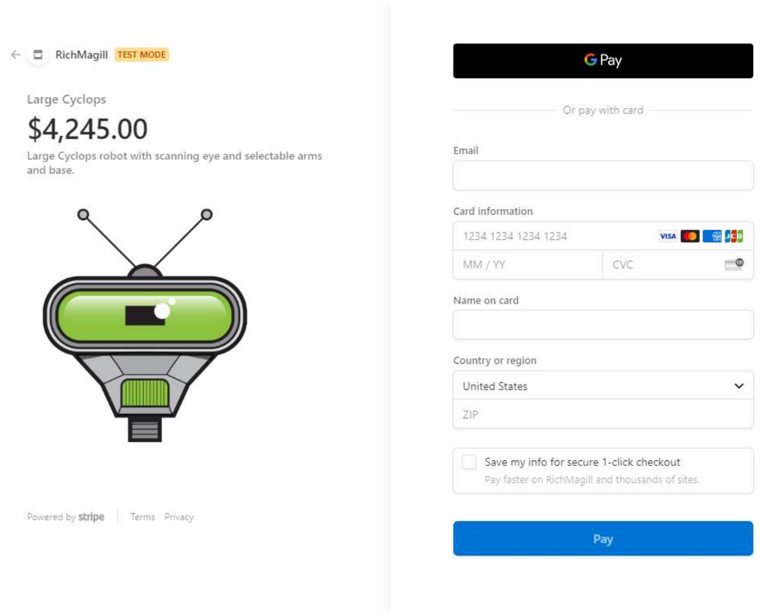
Stripe.com URL.   Note that a test “key” is required to implement Stripe. I got this by creating an account with Stripe. Upon posting to Stripe, the credit card information is processed (in TEST mode), and either a Success (200 OK) or Failure response is received.   Upon receiving the response, Stripe then redirects to the appropriate Success or Cancel URL.

```
// This is a public sample test API key.
// Don’t submit any personally identifiable information in requests made with
this key.
require('dotenv').config();

const stripe = require('stripe')(
  process.env.VUE_APP_STRIPE_KEY,
);

const express = require('express');

// heroku
const serveStatic = require('serve-static');
const path = require('path');

const app = express();

// heroku - here we are configuring dist to serve app files
app.use('/', serveStatic(path.join(__dirname, '/dist')));
// heroku - this * route is to serve project on different page routes except root
`/`
app.get(/.*/, (_req, res) => {
  res.sendFile(path.join(__dirname, '/dist/index.html'));
});

app.use(express.static('public'));
app.use(express.urlencoded());
app.use(express.json());

const SUCCESS = process.env.VUE_APP_SUCCESS_URL;
const FAIL = process.env.VUE_APP_FAIL_URL;

app.post('/create-checkout-session', async (req, res) => {
  let mapped = [];
  if (req.body.price.constructor === Array) {
    mapped = req.body.price.map((price) => ({ price, quantity: 1 }));
  }
  const session = await stripe.checkout.sessions.create({
    line_items: (req.body.price.constructor === Array) ? mapped : [req.body],
    mode: 'payment',
    success_url: `${SUCCESS}?session_id={CHECKOUT_SESSION_ID}`,
    cancel_url: `${FAIL}?canceled=true`,
  });

  res.redirect(303, session.url);
});

const port = process.env.PORT || 8080;
app.listen(port);
console.log(`app is listening on port: ${port}`);
```

Stripe Integration

I created an account in the Stripe application and entered Products and Prices in the stripe dashboard, including all the various robot types (Stripe, NDa). This setup allows Stripe to recognize and accurately price the selected items when the HTTP Post is received.

With the Node server, test key, and products in place, the user is then presented with a Stripe.com checkout page when they select a robot and click “checkout”:


RichMagill TEST MODE

Large Cyclops

$4,245.00

Large Cyclops robot with scanning eye and selectable arms and base.

Powered by stripe | Terms Privacy

G Pay

Or pay with card

Email

Card information

1234 1234 1234 1234

MM / YY   CVC

Name on card

Country or region

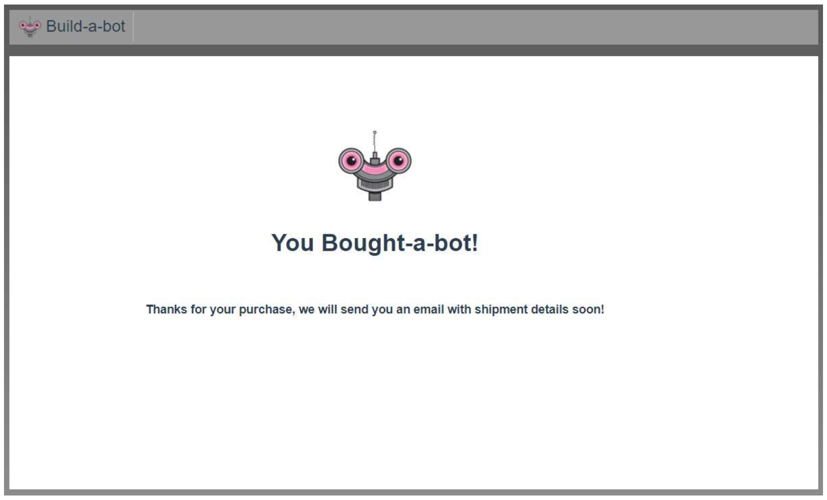
United States

ZIP

Save my info for secure 1-click checkout
Pay faster on RichMagill and thousands of sites.

Pay

For testing purposes Stripe accepts a test “4242 4242 4242 4242” credit card number as a valid card number. Upon success (receiving 200 OK from Stripe), Stripe redirects the user to the success page:

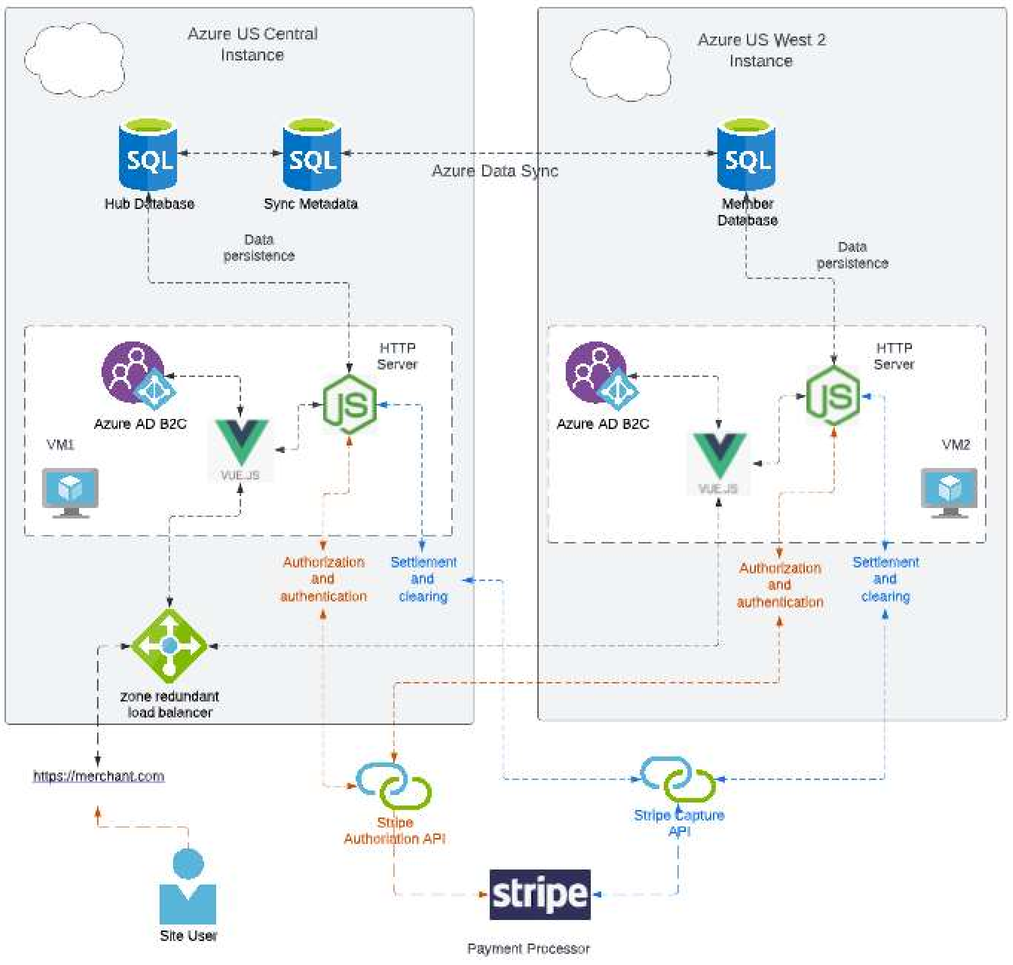


You Bought-a-bot!

Thanks for your purchase, we will send you an email with shipment details soon!

Design with Azure (IAAS)

In my IAAS design, the system is designed for the Azure cloud, implementing Azure AD B2C for identity management, multiple load-balanced Virtual Machines, and replicated Azure database instances. Two nodes are shown in the diagram below, but this is for illustration purposes only. For improved fault tolerance and to prevent the possibility of cascading failure, I recommend at least three nodes of both VM and database instances. While some elements of the design are based on platform services, such as Azure AD B2C, in general I would call this an IAAS design due to the way I have designed the Virtual Machines as infrastructure elements. Here is a diagram:



The recommended VM web servers are DC8 v2 with 8 CPU core and 32GB RAM. The VMs are load balanced with a zone redundant load balancer. The instances should be located in different Azure zones to allow for lower latency and better tolerance to zone failures. The replicated databases are Azure SQL instances, each with 4 CPU core and 32GB RAM. Azure Data Sync would be implemented to sync the databases. The system integrates to the Stripe payment processor via the Stripe Authorization API and the Stripe Capture API.

Azure (IAAS) Cost

To form a basis for the cost analysis, we first begin with an inventory of the general features and resources we plan to implement. The elements to be considered include:

- 3 Virtual Machine Web Servers
- 3 Azure DB instances
- Data transfer charges
- AAD B2C identity service
- Load balancer

These DC8 v2 security-hardened machines are priced at $560.64 per node per month, or $1681.92 monthly far all three nodes (Microsoft, NDa). The database instances are $2290.58 per node per month, or $6871.74 monthly for three nodes (Microsoft, NDb). Leaving out Stripe processing charges, the monthly cost is:

• 3 Virtual Machine Web Servers:	$ 1681.92
• 3 Azure DB instances	$ 6,871.74
• Data transfer charges:	$ 0.00
• AAD B2C identity service	$ 0.00
• Load balancer	$ 1,533.45
• Total	$ 10,087.11

Azure (IAAS) Scalability, Fault Tolerance, and Monitoring

Azure Resource Management (ARM) templates would be created to fully detail and provision the entire system, allowing for rapid deployment of all the infrastructure from the Azure CLI (Microsoft, 2022). While the ARM templates will facilitate setting up the infrastructure, the installation of software and deployment of application instances would be done in a more manual manner. For this reason, scaling the system would most easily be achieved by "scaling up", or adding computing power to the various nodes.

RAM, disk space, and CPU Core can be easily added to scale up the system. Additional memory is relatively inexpensive, but additional CPU Core increase the cost in a more or less linear fashion. For example, increasing the databases from 4 core to 8 core would take the monthly database cost from $6,871.74 to $13,743.48, exactly twice the monthly cost. While expensive, scaling up in this manner can be done on the fly and with minimal engineering effort.

In contrast, “scaling out” in this IAAS design is difficult. Scaling out would involve revising the ARM template for more nodes, redeploying the infrastructure, installing the various software packages, and deploying the application to the new nodes. Continuous Integration and Continuous Deployment (CICD) pipelines would have to be revised and tested as well. While scaling up could be done on the fly, scaling out in this design would likely be a significant engineering project which could take weeks to complete, and certainly could not be done dynamically to deal with increased load or node failures.

For system monitoring in this design, I recommend Azure Monitor (Azure.Microsoft, ND) as it offers a complete suite of tools to monitor applications, network and infrastructure in the Azure cloud. Alerting can be set to automatically notify IT staff in the event of failures, high load, errors, or other events of interest.

With respect to fault tolerance, this IAAS implementation is designed with a high degree of replication and fault tolerance. The selected VMs and databases are highly capable and well provisioned and should be able to handle very high load. Splitting the systems between Azure availability zones provides further assurance that a single failure is unlikely to bring down the entire application.

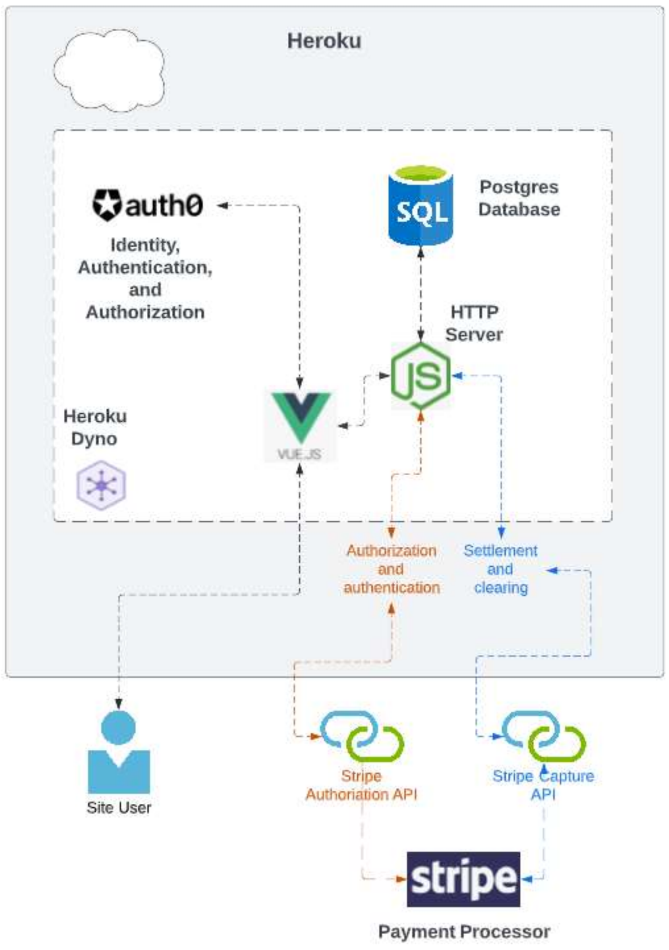
However, this fault tolerance comes at a high cost, in that the system is probably over-designed for the likely load. This is a common issue with IAAS cloud applications, in that companies fear under-provisioned web servers that could paralyze their organization. Accordingly, business leaders often prefer to go with over-designed cloud infrastructure that won't be risky for load-based failure, resulting in tremendous waste. In 2020, for example, it was estimated that companies wasted over 17 billion dollars on unused and over-provisioned cloud resources (Stalcup, 2020). Some estimates indicate that about 35% of all cloud costs are wasted unnecessarily (Weins, 2017).

In order to "right-size" my Production application, reduce costs, and provide for nearly infinite scalability, an approach using containers may be a better fit, especially in light of the fact that I really have no solid baseline to estimate the number of users or load on the system. In other words, I want to be able to start with minimal nodes to start, and grow organically as my user base grows. Use of Docker and Kubernetes was briefly explored but the learning curve was very steep. Exploring other options, I began to settle on Heroku as a viable alternative. The next section of the report covers a Heroku -based PAAS design.

Design with Heroku Containers (PAAS)

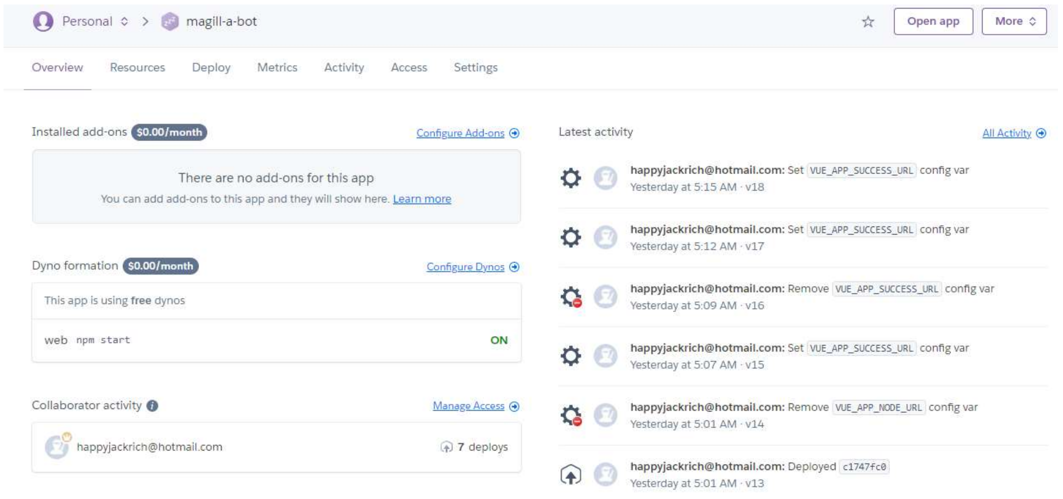
The system design was revised for Heroku. Heroku implements containers known as Dynos. Containers are software packages supplied with all the resources needed for an application to run. They virtualize the operating systems and therefore can be deployed and run virtually anywhere (google, ND). Containers allow for efficient software deployment and are particularly valuable in terms of scalability, since no infrastructure has to be set up prior to deployment. Docker and Kubernetes are well-known as solutions for containerized applications, but similarly well-known is the fact that the learning curve can be steep in deploying apps with this technology (McCarty, 2019).

Looking for alternative solutions that offered a containerized PAAS solution, there are several including Google App engine, Azure App Service, DIgitalOcean, and AWS Elastic Beanstalk (autoidle, 2022). I ended up settling on Heroku as they had good documentation and a free tier that allowed for development of a proof-of-concept at no charge. With Heroku's free tier, the user first deploys the entire application to a single Dyno in a proof-of-concept mode. Once satisfied at that level, the user can then purchase multiple Dynos and can have them provisioned with higher level resources and auto-scaling ability (Heroku, NDa). The following diagram illustrates my design in the Heroku environment.



This illustrates a single node. I can have multiple nodes when I am ready to take that step. Note here that my Vue-Node-Stripe application is essentially the same as it was in the Azure solution, I am just deploying it differently. One other change is that instead of using Azure AD B2C as my Identity provider, I will switch to Auth0.

In order to test out this concept, I deployed my code to my Github repository and then created a Heroku account. After linking the accounts, I created a magill-a-bot project, and added some environment variables.



Installed add-ons $0.00/month

There are no add-ons for this app
You can add add-ons to this app and they will show here. Learn more

Dyno formation $0.00/month

This app is using **free** dynos

web `npm start` ON

Collaborator activity

happyjackrich@hotmail.com 7 deploys

Latest activity

happyjackrich@hotmail.com: Set `VUE_APP_SUCCESS_URL` config var
Yesterday at 5:15 AM · v18

happyjackrich@hotmail.com: Set `VUE_APP_SUCCESS_URL` config var
Yesterday at 5:12 AM · v17

happyjackrich@hotmail.com: Remove `VUE_APP_SUCCESS_URL` config var
Yesterday at 5:09 AM · v16

happyjackrich@hotmail.com: Set `VUE_APP_SUCCESS_URL` config var
Yesterday at 5:07 AM · v15

happyjackrich@hotmail.com: Remove `VUE_APP_NODE_URL` config var
Yesterday at 5:01 AM · v14

happyjackrich@hotmail.com: Deployed `c1747fc0`
Yesterday at 5:01 AM · v13

I was able to successfully deploy the application to the cloud running on a single Dyno. You can run it yourself at https://magill-a-bot.herokuapp.com. Scaling from here is a simple matter of selecting the number and type of additional Dynos and paying for them.

Heroku (PAAS) Cost

The current deployment is for proof-of-concept only and would need to be able to expand to multiple nodes in order to be production ready. One way to calculate how many nodes would be required to match the power of my IAAS design is just to match the RAM provisioned. The IAAS design in Azure had 96 GB RAM in total. At the Perfomance-level Dyno, the Performance L has 14 GB RAM. With that in mind, 6 to 7 Dyno containers should provide a similar level of computing resource (Heroku, NDb).

However, now in this PAAS arrangement we have much greater ability to take advantage of scaling and we can start with fewer nodes since we can easily scale out.   I would recommend starting with 3 Performance L dynos at a cost of $500 each.

Heroku provides either Postgres SQL or Redis (NoSQL) databases.   I would select 3 Postgres SQL databases at a monthly cost of $200 each.

There are other pricing and design options, but with this design, the cloud costs would be in the range of $2100 per month, a huge savings over my IAAS design.   With higher load, and fully scaled to similar RAM to the Azure design, 7 nodes would be approximately $4900 monthly, still half the price of the Azure design.

Heroku (PAAS) Scalability, Fault Tolerance, and Monitoring

Scalability is the real attraction of the Heroku implementation. The application can be manually scaled or auto scaled for load (Heroku, NDc). Scaling is easily set up from the Heroku dashboard.

Monitoring is built into the Heroku environment in the form of numerous indicators integrated into the Heroku dashboard.

Dyno Metrics include Response Time, Throughput, Memory Usage, Dyno Load (threads), Events (such as Deployments, etc.), Errors, Platform Status, Configuration Variable Changes, Scaling Events (adding or removing a Dyno), and any Restarts.

The system can also be configured at application level, and can be integrated to Email, PagerDuty, or Dashboard notifications.

In total, the Heroku environment has more than sufficient monitoring and alerting capability for my needs.

Conclusion

The application was successfully implemented with a Client (Vue) and Server (Node) model that integrates to the Stripe.com API and processing application. The functionality in that respect is very high. Stripe implementation ensures PCI compliance (Stripe, NDb), and further provides styling, validation, error-handling, Luhn algorithm checking, and even fraud detection via Stripe's built-in Radar application. The code is currently implemented with a Stripe "Test" key, and to actually go-live all I would need to do is to replace the test key with a "Production" key.

Cloud deployment with IAAS in the Azure cloud was examined. This design involved deploying the application on three VMs located in different US zones. Redundant Azure database instances in each zone were included in the design as well, and identity was managed via Azure AD B2C. In summary, this design is very robust, but is likely overdesigned for load early in the application lifecycle. It is likely that the application would operate at only a fraction of its capacity starting out. Scaling the IAAS design is perfectly reasonable with respect to scaling up in resources, but difficult with respect to scaling out in nodes. Overall, the monthly cost of this design is projected in the range of $10,000.

Cloud deployment with PAAS in Heroku was also examined and was actually done. This design involved deploying containerized application instances in Heroku Dynos. To begin production operation, I would recommend 3 Performance L Dynos. Manual or Autoscaling is configurable as well such that more nodes can be added as necessary with load, or in the event of a node failure. The use of Heroku Dynos was a much simpler introductory step into containerized design as compared to using Docker and Kubernetes, but a design along those lines is something that should be explored as well. The Heroku solution with 3 nodes has a projected cost in the range of $2100.

Overall, either the PAAS or IAAS approaches to this design would work. However, for reasons of cost, scalability, and ease of deployment, I recommend the Heroku PAAS design as the better solution.

It more thoroughly takes advantage of the cloud environment to deploy the application in a robust and sustainable manner, without the waste of over-provisioned infrastructure resources.

References

Pluralsight.com, ND. *Vue.js Fundamentals by Jim Cooper*. Retrieved May 11, 2022 from https://app.pluralsight.com/library/courses/vuejs-fundamentals/table-of-contents

Masteringjs.io, 2020. *Create a Simple HTTP Proxy in Node.js.* October 30, 2022 from https://masteringjs.io/tutorials/node/http-proxy

Stripe, NDa. *Manage products and prices.* Retrieved May 11, 2022 from https://stripe.com/docs/products-prices/manage-prices

Microsoft, NDa. *Microsoft Azure VM Selector*. Cloud Computing Services. (n.d.). Retrieved April 23, 2022, from https://azure.microsoft.com/en-us/pricing/vm-selector/

Microsoft, NDb. *Pricing Calculator*. (n.d.). Retrieved April 23, 2022, from https://azure.microsoft.com/en-us/pricing/calculator/

Microsoft, 2022. *What are ARM templates?* Retrieved May 11, 2022 from https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/overview

Azure.Microsoft, ND. *Azure Monitor. Full observability into your applications, infrastructure, and network*. (n.d.). Retrieved May 14, 2022, from https://azure.microsoft.com/en-us/services/monitor/#overview

Stalcup, Katy. 2020. *Wasted Cloud Spend to Exceed $17.6 Billion in 2020, Fueled by Cloud Computing Growth.* Business2community.com, Retrieved May 14, 2022. https://www.business2community.com/cloud-computing/wasted-cloud-spend-to-exceed-17-6-billion-in-2020-fueled-by-cloud-computing-growth-02292542

Weins, Kim. 2017. *Where is the $10B in Waste in Public Cloud Costs?* Flexera.com, Retrieved May 14, 2022. https://www.flexera.com/blog/cloud/where-is-the-10b-in-waste-in-public-cloud-costs/

google, ND. *What are containers?* Google.com, Retrieved May 21, 2022. https://cloud.google.com/learn/what-are-containers

McCarty, Scott. 2019. *How to navigate the Kubernetes learning curve.* Opensource.com, Retrieved May 21, 2022. https://opensource.com/article/19/6/kubernetes-learning-curve

Autoidle, 2022. *10 Heroku alternatives to host your app in 2022.* Autoidle.com, Retrieved May 21, 2022. https://autoidle.com/blog/heroku-alternatives

Heroku, NDa. *How Heroku Works.* Heroku.com, Retrieved May 21, 2022. https://devcenter.heroku.com/articles/how-heroku-works

Heroku, NDb. *Heroku Pricing.* Heroku.com, Retrieved May 21, 2022. https://www.heroku.com/pricing

Heroku, NDc. *Heroku Scaling.* Heroku.com, Retrieved May 21, 2022. https://devcenter.heroku.com/articles/scaling

Stripe, NDb. *A guide to PCI Compliance.* Stripe.com, Retrieved May 21, 2022. https://stripe.com/guides/pci-compliance