

# Table of Contents

1. [Professional Self Assessment](#)
2. [Code Review](#)
3. [Plutomi Artifact & Enhancements](#)
4. [Narratives](#)
  - [Software Design and Engineering](#)
  - [Algorithms and Data Structures](#)
  - [Databases](#)
5. [Outcomes](#)

## Professional Self Assessment

During my time at SNHU, I got to experience a wide variety of computer science topics ranging from building web apps, interacting with databases, the software development lifecycle using waterfall and agile methodologies, automation with CI/CD pipelines, client & server development, and the algorithms that power everything under the hood.

As a software engineer, I was already very familiar with these topics from my day to day work. By doing the coursework as well as building this ePortfolio, I was able to strengthen my knowledge of these topics and put the lessons I learned to use in a real world scenario in one of my own projects. Some examples of the changes that will be shown in this ePortfolio include:

- Following industry standard best practices when working in a team such as requiring branch protection on the main branch and at least one approval from another team member before being able to merge new changes
- Having separate environments and infrastructure to test those new changes in a production-like environment before they actually go live to the public
- Implementing a CI/CD pipeline that automatically deploys to each environment saving engineer time and preventing mistakes with manual processes and scripts
- Allowing stakeholders to view the progress of desired features and bugfixes by documenting them publicly on GitHub
- Evaluating tradeoffs between different databases that were most appropriate for my use case and the desired performance SLAs
- Following security best practices by storing environment secrets outside of the codebase that can be swapped without a code change
- Limiting our database to only accept connections from our server instances so they are not openly available to the public
- Practicing tenant data isolation by embedding each tenant ID into the queryable index of each entity
- Rearchitecting a core feature to use a more efficient data structure which would remove a massive limitation for our customers
- Using a hash map with a custom sorting algorithm to efficiently sort the data structure above in  $O(n)$  time instead of  $O(n^2)$
- Adding type safety wherever possible to prevent accessing undefined properties in code using TypeScript as well as comments on each property with JSDoc to give context to the developer on how that property is used

As I progress throughout my career, I would like to be knowledgeable in a variety of areas with a specialization in cloud technologies like Amazon Web Services. The changes made in this ePortfolio reflect my professional goals by making me a well rounded engineer since the changes that were made to my artifact touch on various aspects of computer science and the software application development lifecycle as well as my desired future specialization in cloud computing.

In the next few sections I will introduce [Plutomi \(https://plutomi.com/\)](https://plutomi.com/), an [open source \(https://github.com/plutomi/plutomi\)](https://github.com/plutomi/plutomi) applicant tracking system that I created and have been working on for just over a year. The first section is a code review created at the start of CS-499 outlining planned enhancements that I could make to the project. The next section is a deep dive into the technical enhancements made and how they demonstrate my skills and abilities in the three categories of software design and engineering, algorithms and data structures, and databases. Finally, I will describe how these enhancements helped me achieve the desired course outcomes of CS-499 and my computer science program at SNHU.

## Code Review

At the start of CS-499 at SNHU, I created a code review going over my planned enhancements for Plutomi. In the video I describe what Plutomi is, how it can be used by end customers, the planned enhancements I wanted to make to the project as well as the reasoning behind those enhancements. The video can be viewed [here on YouTube \(https://www.youtube.com/watch?v=k08ZBwK6sBw\)](https://www.youtube.com/watch?v=k08ZBwK6sBw)

## Plutomi Artifact and Enhancements

Note: I am using one artifact for all three categories of this ePortfolio. You can view the old and new versions of the artifact in the `old_plutomi` and `new_plutomi` directories, or viewing the [repository on GitHub \(https://github.com/plutomi/plutomi\)](https://github.com/plutomi/plutomi)

## Software Design and Engineering

### Enhancements

- GitHub Actions CI/CD

I implemented a continuous deployment pipeline using GitHub actions that deploys to Amazon Web Services whenever there is a new push into the main branch of the repository on GitHub. This was done by creating a `deploy.yaml` file in the `.github/workflows` directory with specific commands I would like it to run in order. There is also an option to manually run the workflow from the GUI if

needed by providing the environment you would like to deploy to.

```
67 lines (56 sloc) | 2.07 KB

1  name: Release
2
3  on:
4    push:
5      branches: [main]
6    workflow_dispatch:
7      inputs:
8        environment:
9          description: 'Deployment Environment (production / staging)'
10         type: environment
11         required: true
12       notes:
13         description: 'What are you deploying?'
14         type: text
15
- name: Checkout repo
  uses: actions/checkout@v3
- uses: actions/setup-node@v3
  with:
    node-version: ${{ env.NODE_VERSION }}
- name: Install dependencies
  run: npm i && npm i aws-cdk -g
- name: Print destination
  run: echo "Deploying to staging environment"
- name: Configure aws credentials
  uses: aws-actions/configure-aws-credentials@master
  with:
    aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
    aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
    aws-region: ${{ secrets.AWS_REGION }}
- name: Fixing lambda permissions
  run: npm run docker-fix
- name: Deploying to AWS...
  run: cdk deploy --all --verbose --require-approval never
  env:
```

When to automatically trigger the workflow

If deploying manually, you just need to supply the environment to deploy to

Commands to run

- Environment Secrets

In the action itself, you can supply environment variables to each command with a `env` property. This allows us to change environment variables through GitHub's GUI without requiring a code change. We can manually re-deploy to have these changes take effect. We also have separate secrets per environment, one for staging and one for production.

```
- name: Deploying to AWS...
  run: cdk deploy --all --verbose --require-approval never
  env:
    COMMITS_TOKEN: ${ secrets.COMMITS_TOKEN }
    HOSTED_ZONE_ID: ${ secrets.HOSTED_ZONE_ID }
    ACM_CERTIFICATE_ID: ${ secrets.ACM_CERTIFICATE_ID }
    LOGIN_LINKS_PASSWORD: ${ secrets.LOGIN_LINKS_PASSWORD }
    SESSION_SIGNATURE_SECRET_1: ${ secrets.SESSION_SIGNATURE_SECRET_1 }
    NODE_ENV: production
    NEXT_PUBLIC_DEPLOYMENT_ENVIRONMENT: ${ env.DEPLOYMENT_ENVIRONMENT }
    NEXT_PUBLIC_WEBSITE_URL: ${ secrets.NEXT_PUBLIC_WEBSITE_URL }
    MONGO_CONNECTION: ${ secrets.MONGO_CONNECTION }
```

## Environments

[New environment](#)

You can configure environments with protection rules and secrets. [Learn more.](#)

production

11 secrets



staging

11 secrets



- Pull Request Reviews And Code Owners

I implemented branch protection on the repository to prevent anyone, including myself, from accidentally pushing into the main branch and triggering an unintended deploy. All changes made to the project now have to come from a pull request and must be approved by a CODEOWNER, in this case me. When someone makes a pull request, they will now see the following:

Update README.md #772

Open mazupicua wants to merge 1 commit into plutoni:main from mazupicua:patch-1

Conversation 0 Commits 1 Checks 0 Files changed 1 +2 -0

mazupicua commented now

Example commit

Update README.md Verified 950540f

mazupicua requested a review from joswayski as a code owner now

At least 1 approving review is required to merge this pull request.

Still in progress? Convert to draft

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Code owner review required

Waiting on code owner review from joswayski. [Learn more.](#)

Show all reviewers

1 pending reviewer

Merging is blocked

Merging can be performed automatically with 1 approving review.

Cannot Merge, pending my review

- Type Safety

All entities in the codebase now have types to protect developers from accessing properties that do not exist on the entity in code and causing runtime errors.

```

@Entity()
@Index({ name: 'target_array', options: { target: 1 } })
export class Org extends BaseEntity {
  @Property({ type: 'text', unique: true })
  orgId: string;

  @Property({ type: 'text' })
  displayName: string;

  @Property({ type: 'integer' })
  totalApplicants: number = 0;

  @Property({ type: 'integer' })
  totalOpenings: number = 0;

  @Property({ type: 'integer' })
  totalUsers: number = 1;

  @Property({ type: 'integer' })
  totalWebhooks: number = 0;

  @Property({ type: 'integer' })
  totalQuestions: number = 0;

  @Property({ type: 'array' })
  target: IndexedTargetArray;

  constructor({ orgId, displayName, target }: OrgConstructorValues) {
    super();
    this.orgId = orgId;
    this.displayName = displayName;
    this.target = target;
  }
}

const newOrg = new Org({
  orgId,
  displayName,
  target: [{ id: user.id, type: IndexedEntities.CreatedBy }],
});

newOrg.billingTier

```

Types and default values

any

Property 'billingTier' does not exist on type 'Org'. ts(2339)

View Problem Quick Fix... (Ctrl+.)

- JSDoc



All entities now have descriptions for the indexed target array (explained a bit further down) so developers know what properties they can index on

```
let newStage = new Stage({
  name: GSI1SK,

  (property) target: IndexedTargetArray

  Indexed target array for the stage. Indexed properties are:

  NextStage - @string - If it exists, it's the ID of the stage that comes after this
  stage. Type of IndexedEntities.NextStage

  PreviousStage - @string - If it exists, it's the ID of the stage that comes before
  this stage. Type of IndexedEntities.PreviousStage

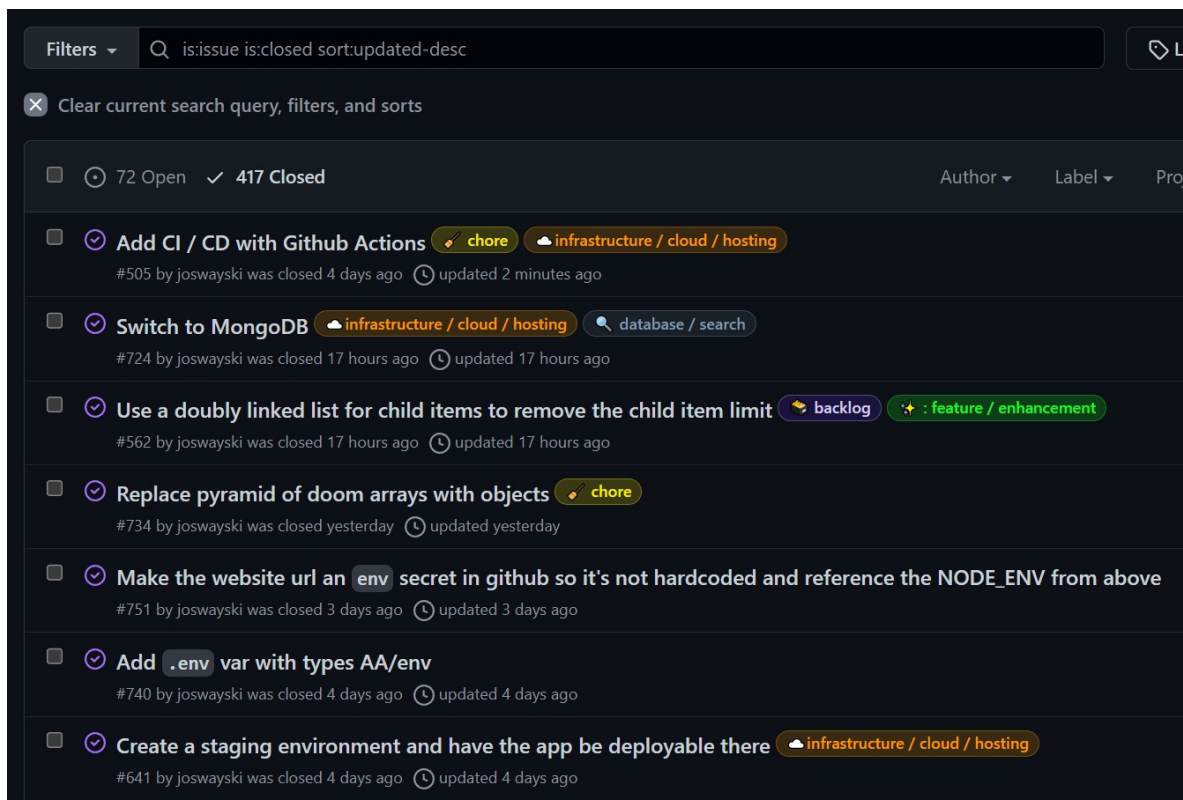
  Org - @string - ID of the org this stage belongs to. IndexedEntities.Org

  Opening - @string - ID of the opening this stage belongs to. Type of
  IndexedEntities.Opening

  target: [
    { id: opening.id, type: IndexedEntities.Opening },
    { id: orgId, type: IndexedEntities.Org },
    { id: undefined, type: IndexedEntities.PreviousStage },
    { id: undefined, type: IndexedEntities.NextStage },
  ],
});
```

- Public Issues

All enhancements were publicly documented in GitHub. Anyone could see what I was working on and my planned enhancements, and if there were any stakeholders they could be informed on what is being prioritized.



## Algorithms and Data structures

- The problem

Since Plutomi is an applicant tracking system, we are allowing our users to create Openings in their organization that people can apply to. These openings have Stages that can be re-arranged whenever the user wants. We were storing the order of these stages on the parent entity (the opening) as an array of stage IDs:

```
const now = Time.currentISO();
const newOpening: DynamoOpening = {
  PK: `${Entities.ORG}#${orgId}#${Entities.OPENING}#${openingId}`,
  SK: Entities.OPENING,
  entityType: Entities.OPENING,
  createdAt: now,
  updatedAt: now,
  orgId,
  openingId,
  openingName,
  GSI1PK: `${Entities.ORG}#${orgId}#${Entities.OPENING}S`,
  GSI1SK: OpeningState.PRIVATE, // All openings are private by default
  totalStages: 0,
  stageOrder: [],
  totalApplicants: 0,
};
```

An issue arises when a user wants to have hundreds, or even thousands of stages in an opening. We would need to retrieve all of these stage IDs *every time we retrieved the parent opening* and on top of that, we now have a theoretical max limit on the number of stages that an opening could have because the opening item in the database would get too big once it crossed a certain threshold (400kb in the case of DynamoDB).

- The solution

Instead of storing stage IDs on the parent, I used a doubly linked list on the stage itself. The stage is now in charge of keeping track of the stage that came before it, and the stage that comes after:



```

{
  _id: ObjectId('634a4f8958971928d7d7a411'),
  createdAt: 2022-10-15T06:13:29.496+00:00,
  updatedAt: 2022-10-15T13:09:10.928+00:00,
  name: "Stage 1",
  totalApplicants: 0,
  totalQuestions: 0,
  target: Array
    0: Object
      id: "634a379d5ecb438ce6297bee"
      type: "Opening"
    1: Object
      id: "beans5"
      type: "Org"
    2: Object
      id: "634a57d958971928d7d7a413"
      type: "PreviousStage"
    3: Object
      id: "634a4f9f58971928d7d7a412"
      type: "NextStage"
  questionOrder: Array

```

**Doubly linked list on a stage**

This removes the stage limit on the openings because you can keep adding stages to it and the item size in the database will stay the same, you'll just create more stage items. This improves performance a ton with the added complexity of having to update multiple items whenever the stage order changes. I will talk more on this below in the [Narratives](#) section along with the sorting algorithm to traverse this doubly linked list very fast.

## Databases

For the database section of this ePortfolio, I migrated Plutomi's database (DynamoDB) over to MongoDB. There are many tiny reasons that eventually cumulated to this decision, but there were two big ones that I have documented below:

1. No adhoc queries

DynamoDB is pretty incredible, boasting over [100 million requests per second](https://aws.amazon.com/blogs/aws/amazon-prime-day-2022-aws-for-the-win/) during their 2022 prime day with single digit millisecond responses. This performance comes at the cost of losing adhoc querying capabilities. Let's take a look at an example. Below is an Opening entity in Dynamo:



```
const newOpening: DynamoOpening = {
  PK: `${Entities.ORG}#${orgId}#${Entities.OPENING}#${openingId}`,
  SK: Entities.OPENING,
  entityType: Entities.OPENING,
  createdAt: now,
  updatedAt: now,
  orgId,
  openingId,
  openingName,
  GSI1PK: `${Entities.ORG}#${orgId}#${Entities.OPENING}S`,
  GSI1SK: OpeningState.PRIVATE, // All openings are private by default
  totalStages: 0,
  stageOrder: [],
  totalApplicants: 0,
};
```

In Dynamo, you would only be allowed to query with the PK and GSI1PK values, and nothing else. These are partition keys (or shard keys depending on who you ask) which allow Dynamo to split the data up across many 10gb storage nodes for what is essentially a hash table but with SSDs. It knows exactly where to go to find your item, because it's broken up by these keys and it doesn't have to scan all of the items in the table.

The two keys I highlighted give you the access patterns of Give me this specific opening by this ID with the PK and Give me all of the openings in this org with the GSI1PK keys. You would then need to do any other type of filtering at your app layer. As I talked to more and more potential customers, they had access patterns that I did not envision which would negate the performance benefits of having the access patterns tightly coupled with the data model, or they simply weren't possible due to the missing adhoc support.

## 2. The 400kb item limit

This was mentioned briefly up above, but Dynamo has an item size limit of 400kb. This is perfectly fine for most practical purposes, but if you would like to embed more data into your item this could fill up fast. I wanted Plutomi to support a virtually unlimited number of stages in an opening and knew someone would hit this limit so it was something I needed to work around. If a user wants to embed applicant responses into the applicant themselves, 400kb might not be enough either. I planned around this by storing applicant responses separate from the applicant and indexing the applicant ID on the response to link them. I also want to give users the ability to add custom metadata to their applicants, and giving them a full megabyte just for that seems more than reasonable.

## How MongoDB addresses these problems

Under the hood, MongoDB and DynamoDB are the same. They both have the same storage partitioning model, they both allow indexing of attributes for faster performance, and they're both extremely fast at these key value lookups which are most of the access patterns in Plutomi:

- Give me all of the users in my org
- Give me all of the webhooks in my org
- Give me all of the openings in an org
- Give me all of the stages in this opening
- Give me all of the evaluation rules in a stage
- Give me all of the questions in the stage
- Give me all of the applicants in a stage
- Give me an applicant by their ID
- Give me all of that applicant's answers to previous questions
- Give me all of the pending invites for my org
- Give me all of the webhooks in my org

As soon as you start introducing a where clause like you would in a relational database, DynamoDB breaks down. It *forces* you to query using an index, but you also can't index every attribute as that will increase your storage costs dramatically.

MongoDB does not force you to use an index lookup allowing for adhoc queries. If you do use indexes though, you can index *arrays* and *json* allowing for multiple properties indexed at once like the picture below. It is common practice to name this property something inconspicuous like *target*, as it's storing pointers to other documents in separate collections or Enum values. Below is an example of an Opening in MongoDB, with it's Org and OpeningState indexed in this target array.

# staging.opening


Documents

Aggregations

Schema

Explain P

 FILTER { field: 'value' }

 ADD DATA ▾



VIEW



```
_id: ObjectId('634a379d5ecb438ce6297bee')
createdAt: 2022-10-15T04:31:25.400+00:00
updatedAt: 2022-10-15T06:56:21.569+00:00
name: "NYC"
totalApplicants: 0
totalStages: 21
▼ target: Array
  ▼ 0: Object
    id: "beans5"
    type: "Org"
  ▼ 1: Object
    id: "PRIVATE"
    type: "OpeningState"
```

You might have noticed that a lot of entities have an OrgId built in to one of these indexes. This not only allows for quicker queries when searching as the query planner will use this index to filter out documents *not* in the org first, but it serves as an extra layer of security due to explicit tenant isolation. This OrgId comes from the user's session, so by default, they will *only* have access to Openings, Stages, Applicants, etc. that are in their org. It is *impossible* for them to get access to another tenant's information by doing some sort of SQL injection.

I also have staging and production databases depending on the deployment environment of the GitHub action:

# Database Deployments

 Find a database deployment...

production

Connect

View Monitoring

Browse Collections


...

R 1.5

W 0.3

Last 6 hours


1.6/s



Connections 74.0

Last 6 hours

79.0



VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS
5.0.13	AWS / N. Virginia (us-east-1)	M10 (General)	Replica Set - 3 nodes	Active

staging

Connect

View Monitoring

Browse Collections


...

R 1.0

W 0.3

Last 6 hours


1.0/s



Connections 68.0

Last 6 hours

85.0



VERSION	REGION	CLUSTER TIER	TYPE	BACKUPS
5.0.13	AWS / N. Virginia (us-east-1)	M10 (General)	Replica Set - 3 nodes	Active

My servers are connecting to the database with a username and password, however, even if those credentials get leaked nobody will be able to access it because I am limiting the connections to only come from the IP addresses of my servers:

You will only be able to connect to your cluster from the following list of IP Addresses:

IP Address	Comment	Status	Actions
52.90.23.202/32	STAGING SERVER 1	Active	<a href="#">EDIT</a> <a href="#">DELETE</a>
67.81.145.7/32 (includes your current IP address)	MY IP	Pending	<a href="#">EDIT</a> <a href="#">DELETE</a>
3.90.17391/32	PRODUCTION SERVER 1	Active	<a href="#">EDIT</a> <a href="#">DELETE</a>

Another bonus is that the MongoDB item size limit is 40x higher than Dynamo's so we have a lot of flexibility there for any use case that might arise.

## Narratives

In the code review video I give some background as to why I decided to build Plutomi. For those that are not able to watch it, I worked on recruiting at a large food delivery company in the US. The system that we used at the time allowed us to scale up quickly when we were starting, but as we grew, we started hitting the limitations of the tool in the form of API throttling, long load times, and slow iteration speed from the vendor despite us being their largest customer by far.

It would have really benefited us to have an open source solution that our developers could contribute to if needed to implement the changes that we desired, as well as to host our own solution if we wanted to go that route. This is why I set off to build my own version to address these shortcomings.

The reason it was chosen for this ePortfolio, as well as for each category of improvement, is because this is a very large project that had room for improvement in every category due to hindsight of past design decisions, customer feedback, hitting limits of our own due to our infrastructure, and many manual steps that should have been automated from the start.

## Software Design and Engineering

Whenever I made a change to the project, I would make a pull request, review it myself, merge and deploy. The deployment step was a manual process: I would run the `cdk deploy` command with production credentials from my laptop and pray for the best. This had many downsides with the obvious one being having to manually do this every time I wanted to make a change, but it was also extremely prone to failure. If my laptop went to sleep, the deployment would crash. There was no production-like environment to actually test the changes, it was only development or live code that people are using. Some tools that *I was using* also behave very differently when the `NODE_ENV` is in development or production like Nextjs, and I had no real way to test this either. I knew I had to automate this somehow and luckily I had experience with GitHub actions from my job and was able to quickly setup a deployment script and this process was automated relatively painlessly.

Another side effect of implementing the deployment pipeline above is that, if we ever want to create a new environment, this is as simple as changing the `DEPLOYMENT_ENVIRONMENT` variable in GitHub and re-deploying. If someone makes a pull request, I can have the same action kick off and deploy to an environment with their GitHub username suffixed by their branch name, giving them a playground to test their changes without them having to setup any infrastructure, or know how any of it works.

Since this project is open source, I eventually expect people to contribute to the project in a small capacity. With the mandatory code reviews implemented using the CODEOWNERS feature of GitHub, I can ensure that any changes *must* be reviewed by someone before being merged. If I'm ever away, I can add a member of the community who's contributed before as a CODEOWNER and they can take the reigns while I'm on vacation without giving them access to my GitHub account.



Creating issues in the public repository also allows a crude road map for everyone to see the current state of the project as well as planned features, and if they are currently being worked on or not with the labels on them. It also helps to keep you organized since you can link issues with other issues if one depends on another by writing Blocked by #147, and GitHub will automatically create a comment on #147 saying that this issue referenced it. You can also close these issues automatically when a PR is merged by writing in the PR notes Closes #147 or Fixes #147. This whole issue creation might seem tedious since I am the only one working on this at the moment, but it helps to treat it like a real job as it's not very different from my actual day to day job with all the formality.

By adding types, I ensure that developers can leverage the features of their IDE for auto complete as well as preventing runtime errors since the app will not run if the types are mismatched. You cannot say that a newly created Opening has a NextStage property unless it is specifically defined in the entity. By adding JSDoc comments, any developer knows what each property type is, as well as the accepted values without having to leave their codebase.

Since our app is Dockerized, running inside a [Fargate \(https://aws.amazon.com/fargate/\)](https://aws.amazon.com/fargate/) task, but also running a Nextjs app, environment variables were tricky. You essentially have three sets of environment variables:

1. Those running in the GitHub action that can be passed to the cdk deploy command
2. Those running in the Docker container for the app that has the API for things like database secrets or login link signatures
3. Those running on the front end *only*, for Nextjs

It was very painful figuring out what needed access to which credentials as I was using a package called simple-env which added type safety to environment variables, however it did not make them available at build time for our Nextjs app. This took a while to debug, and it was compounded by the fact that if I needed to test a deployment to see if it worked, I would have to wait the full 12 minutes for AWS to spin up all of the required infrastructure. I settled on creating my own environment variable Type, and sharing that with the front end for Nextjs. All being said, it is extremely helpful being able to change the variables from the GitHub GUI and not have to touch any code.

## Algorithms and Data Structures

This was without a doubt the most fun and challenging part of all of the enhancements. By implementing the doubly linked list for stages in an opening mentioned above, this brought with it its own set of problems.

1. When deleting a stage, we now need to update the previous stage's NextStage property to be the deleted stage's NextStage property AND the next stage's PreviousStage property to be the deleted stage's PreviousStage property. Two if checks, not too bad!
2. When adding a stage, by default, we add it to the end. We simply get the last stage using `allStages[allStages.length - 1]?.id` ?? undefined to get the last stage's ID, set it as the PreviousStage, and NextStage is undefined. The same method can be used to put a stage anywhere in the middle, just find the previous and next stages respectively.

## NYC - Settings

 / Applicants / Opening Settings

+ Add Stage

### Stage Order

Stage 1

 0

Stage 2

 0

Stage 3

 0

Stage 4

 0



This is a *must have* feature for Plutomi. When stage IDs were stored in an array in the opening, reordering them was extremely easy as it was moving an array item down or up a few places. With doubly linked lists, things get very difficult. First, you have to account for all of the edge cases that you may encounter... and there are 16 in total:



Writing code to handle each of these specific edge cases would get repetitive extremely quickly, so I implemented a way to check for scenarios that repeat themselves and essentially group the updates when they overlap on a stage. But first, to even get to this point, we need our stages to be

sorted. If we're going to be re-ordering stages, we need to know the correct order before we move our stages around.

The sortStages algorithm that I implemented takes what would be a  $O(n^2)$  sorting algorithm down to  $O(n)$ . The naive approach for sorting a doubly linked list is to:

1. Find the first stage in the list by checking if the PreviousStage is undefined
2. Finding the second stage in the list by traversing the list again and finding the next stage that has the first stage's NextStage ID
3. Keep traversing the list until all stages are found and sorted

You could have a scenario where each subsequent traversal has the next stage at the end of the list with horrible performance. The way I solved this is by:

1. Traversing the list once. Push all stages that are *not* the first stage into a hash map with the stage ID as the key and the stage itself as the value
2. If we found the first stage, we push it into an array called sortedStages
3. Once we've added all of the stages to our hash map, we need to recursively loop once more starting with the first stage from earlier, getting the nextStage which is now a  $O(1)$  query in our hash map, and push it into the sorted stages array.

```
let reachedTheEnd = false;
let startingStage = firstStage;

while (!reachedTheEnd) {
  const newNextStageId = findInTargetArray({
    entity: IndexedEntities.NextStage,
    targetArray: startingStage.target,
  });

  let nextStage: Stage | undefined;

  if (newNextStageId) {
    nextStage = mapWithStages[newNextStageId.toString()];
    sortedStages.push(nextStage);
    startingStage = nextStage;
    // Continue loop until all stages are sorted
  } else {
    reachedTheEnd = true;
    break;
  }
}

return sortedStages;
```

This greatly improves the performance at scale due to the minimal array traversals that we have to make. Now back to re-ordering stages. Say we have three stages in order: 1, 2, and 3. If we moved stage 1 to be in the middle between stages 2 and 3, what has *changed*?

- Stage 1's PreviousStage is now Stage 2
- Stage 1's NextStage is now Stage 3
- Stage 2's PreviousStage is now undefined
- Stage 2's NextStage is now Stage 1
- Stage 3's PreviousStage is now Stage 2

You can see how we can bucket the 16 different edge cases above into four basic conditional checks:

1. There was a stage behind us before we moved
  - Needs its PreviousStage updated? Y/N
  - Needs its NextStage updated? Y/N
2. There was a stage after us before we moved
  - Needs its PreviousStage updated? Y/N
  - Needs its NextStage updated? Y/N
3. There is a new stage behind us after we moved
  - Needs its PreviousStage updated? Y/N
  - Needs its NextStage updated? Y/N
4. There is a new stage after us after we moved
  - Needs its PreviousStage updated? Y/N
  - Needs its NextStage updated? Y/N

I added comments to the code snippets below so anyone else can picture these scenarios without having to go to the literal drawing board. Here is the code for checking the first two conditions, and the same can be done on the *after we moved* checks. I recommend [viewing the code directly](https://github.com/plutomi/plutomi/blob/main/Controllers/Stages/updateStage.ts#L88-L331) (<https://github.com/plutomi/plutomi/blob/main/Controllers/Stages/updateStage.ts#L88-L331>) for clarity.



```

// If there was a previous stage before we moved, we want to update that stage
if (oldPreviousStageId) {
  oldPreviousStage = allStagesInOpening.find((stage) => stage.id === oldPreviousStageId);

  oldPreviousStagesNextStageIndex = oldPreviousStage.target.findIndex(
    (item) => item.type === IndexedEntities.NextStage,
  );

  /**
   * Set the old previous stage's nextStage to be the old next stage of the stage we are currently moving
   *
   * OLD --- NEW
   * Stage 1 --- Stage 1 <-- Its next stage property gets updated to Stage 3, which was Stage 2's old next stage
   * Stage 2 --- Stage 3
   * Stage 3 --- Stage 2 <-- Moved
   */
  oldPreviousStage.target[oldPreviousStagesNextStageIndex] = stage.target.find(
    (item) => item.type === IndexedEntities.NextStage,
  );
} else {
  /**
   * Set our old next stage's previous stage to be undefined.
   *
   * OLD --- NEW
   * Stage 1 --- Stage 2 <-- Previous stage is now undefined
   * Stage 2 --- Stage 1 <-- Moved
   *
   * We can't do the update here because we don't know if that next stage exists yet - we are doing that down below so...
   * to prevent duplicate checks for that next stage, we are setting a reminder for us to update that stage later once we verified that it exists
   */
  updateOldNextStage = true;
}

// If there was a next stage before we moved, we need to update that stage's previous stage
if (oldNextStageId) {
  oldNextStage = allStagesInOpening.find((stage) => stage.id === oldNextStageId);

  oldNextStagesPreviousStageIndex = oldNextStage.target.findIndex(
    (item) => item.type === IndexedEntities.PreviousStage,
  );

  /**
   * We need to set Stage 2's previous stage to our stage's old previous stage
   *
   * OLD --- NEW
   * Stage 1 --- Stage 2 <-- Previous stage gets updated to undefined
   * Stage 2 --- Stage 3
   * Stage 3 --- Stage 1 <-- Moved
   */
  oldNextStage.target[oldNextStagesPreviousStageIndex] = stage.target.find(
    (item) => item.type === IndexedEntities.PreviousStage,
  );
} else {
  /**
   * If there is no old next stage, we need to update our old previous stage's next stage ID to be undefined.
   *
   * OLD --- NEW
   *
   * Stage 1 --- Stage 2 <-- Moved
   * Stage 2 --- Stage 1 <-- Needs their next stage to be undefined
   *
   * Same case as above, we don't really know if we had a previous check, this logic statement is not responsible for that.
   * We are setting a reminder for us to check it a little bit below to prevent duplicate logic
   */
  updateOldPreviousStage = true;
}

```

```

// Update the relevant stages if needed (the two else checks above)
if (oldPreviousStage && updateOldPreviousStage) {
  oldPreviousStage.target[oldPreviousStagesNextStageIndex] = {
    id: undefined,
    type: IndexedEntities.NextStage,
  };
}

if (oldNextStage && updateOldNextStage) {
  oldNextStage.target[oldNextStagesPreviousStageIndex] = {
    id: undefined,
    type: IndexedEntities.PreviousStage,
  };
}

// Queue them up to be saved into the DB
entityManager.persist(oldPreviousStage);
entityManager.persist(oldNextStage);

```

## Databases

As stated above, the main reason for my switch from DynamoDB to MongoDB was for the adhoc querying functionality that was badly needed to support future use cases. We still get the same great performance of Dynamo, although with a bit more overhead as there are actual servers to worry about now and *it is* a bit pricier. The APIs of each are a LOT different. There are no good ORMs for Dynamo, you essentially have to make your own. Type safety is an afterthought, and god forbid you use one of the [many reserved words](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ReservedWords.html) (<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ReservedWords.html>) as a property name.

We are limiting access to the database via IPs, and embedding a tenant ID right into our queries and indexes for performance and data isolation.

## Outcomes

1. By having public issues on GitHub for everyone to see, comment, edit and collaborate on, adding main branch protection rules to ensure that all changes are reviewed as part of a team, and creating separate environments specifically for staging that require a manual approval to push to production, I employed strategies for building collaborative environments that enable diverse audiences to support organizational decision making in the field of computer science.
2. By adding type safety to entities, inline comments to show the logic behind the doubly linked list stage updating logic, as well as JSDoc comments which allow developers to see the inputs and outputs of a function right in their code editor, and creating publicly available issues on GitHub with outlined steps on what will it will take to accomplish each task, I designed, developed, and delivered professional-quality oral, written, and visual communications that are coherent, technically sound, and appropriately adapted to specific audiences and contexts.

3. By migrating the stages to use a doubly linked list to maintain their order in the parent entity allowing for an unlimited number of stages to be created per opening and creating a sorting algorithm to traverse the doubly linked list and sort it in a reasonable time, I designed and evaluated computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution, while managing the trade-offs involved in design choices.
4. By implementing an automated continuous delivery pipeline which deploys changes to a live environment to get rapid feedback and improve iteration speed in an agile environment, and migrating databases to MongoDB to support ever changing access patterns from customers, I demonstrated an ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals.
5. By forcing database queries to use a tenant ID, locking connections to the database to come from a specific set of IP ranges, and storing secrets outside of the codebase, I developed a security mindset that anticipates adversarial exploits in software architecture and designs to expose potential vulnerabilities, mitigate design flaws, and ensure privacy and enhanced security of data and resources.