hoagieplan: Princeton Academic Planning Reimagined

Windsor Nguyen
windsor@princetonalignment.org

Abstract

The current ecosystem of outdated and fragmented apps presents challenges in maintenance, user experience, and adaptability to the evolving academic landscape at Princeton University. We introduce `hoagieplan`, an open-source application that aims to revolutionize the academic course planning experience for Princeton University students by consolidating the functionalities of existing TigerApps such as TigerPath, ReCal, and PrincetonCourses into a single, modern web application. This paper focuses on the specifics of the weekly calendar scheduling portion of `hoagieplan` and also outlines additional stretch goals for the application.

1 Introduction

The majority of Princeton undergraduate students use one or more of the "TigerApps trinity":

1. TigerPath[1]
2. ReCal[2]
3. PrincetonCourses[3]

These web applications, part of a broader ecosystem known as *TigerApps*[4], were written many years ago with the ReCal being a decade old. This presents several problems for its users.

Firstly, the maintenance of these web applications requires additional undue labor in order to cater to outdated technology stacks. Second, having three separate applications for the course planning process disrupts what ought to be a seamless and efficient experience hosted on a single platform. Course planning should not require a student to have 10+ tabs open across several different fragmented *TigerApps* in order to effectively gather all the information they need. Lastly, Princeton students deserve to experience the offerings of modern web applications to the fullest extent, not buggy decade-old technology.

This paper introduces `hoagieplan`[5], an academic planning app that aims to seamlessly allow Princeton University undergraduates to research and plan for their academic futures by improving upon the primary features of the TigerApps trinity, built with a cutting-edge technology stack. The goal of `hoagieplan` is to serve as a paragon for modern academic course planning for Princeton undergraduate students.

This paper will focus on the app's weekly calendar planning portion, henceforth known as the "hoagieplan schedule," as well as a novel integration of a large language model that was finetuned

[1] `https://tigerpath.io`, a semesterly course planning tool with requirements checking functionality.
[2] `https://recal.io`, a weekly course planning tool.
[3] `https://princetoncourses.com`, used to explore course listing details.
[4] `https://tigerapps.org`, the main web applications hub for Princeton students.
[5] `https://plan.hoagie.io`

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

on a dataset of Princeton course data consisting of 300,000+ examples.

1.1 Related Work

TigerPath, ReCal, and PrincetonCourses represent significant achievements in student-led software engineering at Princeton. Yet, the academic landscape at Princeton continues to evolve, such as with the introduction of minor programs[6]. Moreover, TigerPath's management by the TigerApps team has waned[7]. Additionally, departmental requirements frequently change. For instance, TigerPath has not been updated to include the new *COS 240* requirement for computer science majors or to reflect the department name change from "ELE" to "ECE" for Electrical and Computer Engineering. Other inaccuracies, such as the misspelling of the econometrics requirement for the economics major as "Economometrics," contribute to a perception among students that TigerPath, despite its potential, has become outdated and unreliable. Indeed, students should not have to verify the correctness of an application that is intended to provide accurate academic planning.

Moreover, the modern Princeton student should not have to settle for software that was written nearly a decade ago. PrincetonCourses was first released six years ago in the spring of 2017. TigerPath was released a year later in the spring of 2018 and was influenced by PrincetonCourses, opting to delegate the course details portion of the course planning experience as outbound redirecting links to PrincetonCourses. The oldest of the three, ReCal, was the result of an independent work project from the fall 2014-15 academic year, nearly a decade ago!

There have been several attempts at unifying the fragmented course planning ecosystem. A non-exhaustive list of such attempts include the student design group ResInDe's[8] "Course Selection" project, TigerJunction[9], and ReCal++ which is a defunct COS 333 project from the 2022-23 academic year.

ResInDe's "Course Selection" was an ambitious project that attempted to bridge a similar gap in the course planning ecosystem as `hoagieplan`. Unfortunately, it appears that the project is no longer actively managed as the last commit to their GitHub repository[10] was two years ago.

TigerJunction was an independent effort from Joshua Lau '26 outside of COS 333 and has arguably been the best attempt so far at unifying the course selection ecosystem. It was publicly announced in the middle of the fall 2023 semester as Compass was being developed in COS 333. In its current state, TigerJunction can be viewed as a more modern ReCal with an expanded feature set such as changeable themes and additional filters. Before being incorporated into the TigerApps ecosystem, TigerJunction had announced plans to expand their functionality suite to include a TigerPath-like semesterly planning functionality as well as a tool to discern course prerequisites, goals that are very similar to what `hoagieplan` offers.

Another notable TigerApp is *TigerSnatch*[11] by Shannon Heh '23, Nicholas Padmanabhan '23, and Byron Zhang '23. TigerSnatch allows students to be notified of open slots in lectures and precepts and has proven to be useful to many students during the add-drop period, often boasting hundreds of users at any given time subscribed to be alerted of a course roster change. TigerSnatch is an emerging cornerstone of the Princeton web application ecosystem, but such a functionality is not a top priority of the `hoagieplan` project in our current state and therefore will not be discussed in this paper.

[6] Faculty Approve Academic Minors at Princeton
[7] As of March 2024, the last significant update to the TigerApps GitHub repository occurred over a year ago.
[8] `https://www.princetonresinde.com/work/course-selection`
[9] `https://junction.tigerapps.org/`
[10] `https://github.com/PrincetonResInDe/course-selection`
[11] `https://tigersnatch.com`

1.2 Context

`hoagieplan`, originally named `Compass`, was originally developed as a COS 333 project in fall 2023 by George Chiriac '25, Julia Kashimura '25, Ijay Narang '25, Kaan Odabas '25, and Windsor Nguyen '25.

In its current state, Compass is a web application for Princeton students to explore courses and to plan their four-year academic schedule. Users can select their major and up to three intended minors. Then, they can search for courses, read their descriptions and reviews, and place them in their schedule. The application displays which major and minor requirements are satisfied. It also displays which requirements are not satisfied, and lists the courses they require.

While Compass is already a useful application, we believe that it is far from our vision for it — the all-in-one academic planning application that will replace TigerPath, ReCal, and PrincetonCourses. At the end of the fall 2023 term, Compass was a great web application with PWA (Progressive Web Application) capabilities but ultimately fell short of its expansive unifying goals. The original team delivered Compass as a minimum viable product, boasting semesterly planning (TigerPath) functionality but with minimal course researching capabilities (PrincetonCourses) and no calendar (ReCal) feature. To this end, we plan to enhance Compass by implementing — and improving upon — the calendar feature as the primary focus of this Independent Work project.

The application is named `hoagieplan` because it is hosted on the `hoagie.io`[12] platform.

2 Features

This section describes the primary features and functionality of `hoagieplan schedule`. Note: For a detailed view of the descriptions, please refer to the Appendix.

2.1 Calendar Grid

`hoagieplan` offers a neat and tidy calendar grid spanning Monday through Friday from 8:00 AM to 9:00 PM to capture the full range of potential class activity. This part took much longer than expected to complete and was highly non-trivial to code from scratch.

2.2 Semester Pagination

Students may choose to plan out the weekly calendar view going as far back to Fall 2020 and one semester in advance, e.g. Fall 2024 after the halfway mark in the Spring 2024 semester. Students may also choose up to 5 scheduling configurations per semester, i.e. five distinct sets of schedule combinations for a given semester.

2.3 Widgets

2.3.1 Search Filters

Students may find it useful to narrow their search down using filters. The dominant calendar app on campus, ReCal, does not offer this functionality. Next to `hoagieplan calendar`'s search bar is a small settings widget that allows the student to filter by *Distribution Area*, e.g. Social Analysis, *Course Level*, e.g. 400-level courses only or any combination from 100- to 500-level courses, *Allowed Grading*, e.g. PDF-only classes.

2.3.2 Recent Searches

Students may find it useful to re-add a course that they had previously removed from their schedule. Under the search bar, there are oval buttons with the student's most recent valid searches. Since these are cached, the student will have $O(1)$ access to the previously searched course.

[12] `https://hoagie.io`

2.4 Selected Courses

Students can see which courses they've selected in a neat tabular column. To the right side of the course card is an ergonomic "Remove" button for the student to easily adjust their schedule.

2.5 Schedule

`hoagieplan schedule` allows the student to search for and resolve conflicts between courses from across 11,000+ different courses and over 1,000+ in any given semester.

2.5.1 Conflict Resolution

If two or more courses coincide, then they divide their . In general, if $\mathcal{N}$ or more courses coincide, then the app looks for how many other conflicting courses coincide with a given course.

Suppose a student had course $\mathcal{A}$, which runs from 2:00 to 4:20 PM, course $\mathcal{B}$, which runs from 1:30 to 2:50 PM, and course $\mathcal{C}$, which runs from 3:00 to 4:20 PM, and suppose the time frame $\mathcal{T}$ was between 1:00 to 4:30 PM. A naive approach to solve this problem would be to take all conflicting courses within a given time frame, in this case all three courses, and divide their widths by 3. In practice, this is not optimal as course $\mathcal{B}$ and course $\mathcal{C}$ do not conflict but would both have only a third of their original width regardless.

More generally, consider a given set of $\mathcal{N}$ courses within some arbitrary time frame $\mathcal{T}$ on the calendar grid, `hoagieplan schedule` will choose the course $\mathcal{C} \in \mathcal{T}$, and suppose it has $\mathcal{K}$ other conflicting courses, that has the *least* amount of conflicting courses to operate on. In this case, `hoagieplan schedule` divides the width of $\mathcal{C}$ by $\mathcal{K}$.

3 Technology Stack

The section dives into the specifics of `hoagieplan`'s technology stack.

3.1 Frontend

`hoagieplan` opts to use Next.js as our frontend framework. Next.js is trusted by OpenAI, Anthropic, Meta, Under Armour, Nike, Stripe, Spotify, and countless more that require robustness and reliability at scale. There are countless reasons for why we view Next.js as the ultimate web development framework but it boils down to three reasons:

1. **Easy hosting.** Next.js is owned by Vercel and has extremely dedicated and highly reliable hosting. Behind the scenes, Vercel provides us with important insights such as detailed analytics about visitors, page views, browser/device, as well as reports about the app's speed performance across all users.
2. **Hybrid Rendering Options.** Next.js excels with its versatile rendering capabilities, offering:
   - **SSG (Static Site Generation)**: Pre-renders pages at build time, enhancing load times and SEO performance.
   - **SSR (Server-Side Rendering)**: Renders pages on each request, ideal for handling dynamic content.
   - **ISR (Incremental Static Regeneration)**: Combines SSG and SSR by allowing pages to be updated incrementally after deployment without needing a full rebuild.

   This flexibility allows developers to choose the most suitable rendering strategy for different scenarios, optimizing performance across all user experiences.
3. **Built-in Routing and API Routes.** Next.js simplifies page routing and API management by:
   - Utilizing a file-based routing system where the filesystem is used as the API, automatically converting file names in the `App` directory into URLs.
   - Allowing the creation of API routes to handle server-side logic directly within the same project, reducing the complexity of developing separate backend services and facilitating a more integrated approach to building web applications.

3.1.1 Styling

Our styling strategy primarily revolves around Tailwind CSS. Tailwind CSS is a type of tool used to design websites, known as a "CSS framework." It provides ready-to-use building blocks, called utility classes, which you can use to style your website directly within your webpage's code (HTML). This approach allows for quick customization and a consistent look across the website, minimizing extra coding by removing unnecessary styles automatically.

In addition to Tailwind CSS, we incorporated styles from other frameworks like Material UI, Daisy UI, and Evergreen UI. These are collections of pre-designed buttons, menus, and other elements that help us quickly create an attractive and functional website. Each framework has its unique style and components, which means we can choose the best features from each to suit our project's specific needs.

Furthermore, we used something called Sass CSS for the more complex styling tasks. Sass, short for Syntactically Awesome Style Sheets, is an enhancement over traditional CSS (the standard language for designing web pages). Sass allows us to write CSS code in a more structured and efficient way. It offers features like variables (to reuse certain styles without repeating code), nested rules (to organize CSS rules in a hierarchical structure), and mixins (to group CSS declarations that can be reused throughout the website). These features make the styling code easier to work with and understand, especially when dealing with larger stylesheets or more dynamic, interactive web pages. This is particularly beneficial over standard CSS, which can become cumbersome in complex scenarios.

3.1.2 State Management

`hoagieplan` uses Zustand for our state management. In web development, "state" refers to the storage and management of data that determines the behavior and appearance of a web application at any given time. Managing state effectively is crucial for ensuring that the app responds correctly to user interactions and other changes.

Zustand is chosen for its simplicity and efficiency over larger, more complex systems like Redux, another state management tool. While Redux is powerful, it often requires a lot of additional code ("boilerplate") to handle simple tasks. Zustand streamlines this process, providing a straightforward way to manage state without the extra complexity and is also incredibly robust, handling a wide variety of tasks reliably.

We use "Zustand stores" to hold and manage our application's state. These stores are containers that hold data and allow different parts of our application to access and react to this data as needed. Additionally, we use Zustand's `(persist)` API, which helps in saving the application's state in a way that it can be restored even after the app restarts or the page reloads.

Zustand also offers several other impressive features:

- **Immutability with a mutable API**: You can write code as if you are directly modifying the state, but Zustand ensures that these modifications do not affect other parts of the state unexpectedly.
- **Hooks integration**: Zustand works seamlessly with React hooks, making it easier to integrate and use state in React components.
- **Minimal re-rendering**: Zustand only triggers re-renders of components when absolutely necessary, which improves performance.
- **Custom middleware support**: You can extend Zustand with custom functionalities, tailoring the state management to fit specific needs of the application.

- **DevTools integration**: Zustand can be connected to developer tools for easier debugging and state tracking.

These features make Zustand a flexible and powerful choice for efficiently managing state in modern web applications.

3.2 Backend

We use Django for our backend, a framework that's famously designed for "perfectionists with deadlines." This characterization perfectly fits the bill for our student project, which is typically constrained to a single semester. Django is built on Python and provides a suite of powerful tools that streamline backend development.

This framework allows for quick and intuitive creation of API endpoints through its elegant URL routing system. The system is straightforward, enabling developers to efficiently map URLs to appropriate views, a crucial aspect for a seamless user experience. Django's comprehensive model system also interacts flawlessly with our chosen database, PostgreSQL, ensuring that data handling is both efficient and robust.

Further, Django integrates seamlessly with Heroku, our hosting service, which simplifies deployment and scalability challenges. Opting for Django means we can develop rapidly without sacrificing functionality or quality, making it an ideal choice for projects with tight timelines but high standards.

3.2.1 Hosting Service

We use Heroku as our hosting service, and the reasoning behind this choice is two-fold:

1. **Reliability and Cost-Effectiveness:** Heroku is known for its reliable performance and cost-efficiency, especially for hosting Django applications. Its platform supports quick deployments and scalable solutions, which are crucial for our developmental pace and budget constraints.
2. **Community Preference and Additional Benefits:** Heroku is a popular choice among TigerApps at Princeton, making it a familiar and trusted option. When we started in COS 333, we aimed for a solution that would not only serve us in the short term but also integrate well into the existing technical framework. Additionally, the availability of Heroku credits through the GitHub Student Developer Pack was a compelling perk, allowing us to leverage more advanced features at a reduced cost.

Choosing Heroku aligns with our long-term planning and ensures that we are integrating with the broader application ecosystem at Princeton. It also allows us to maximize our resources, leveraging deals such as the GitHub credits, which are particularly beneficial for students.

3.2.2 Authentication

CAS is the authenticator of choice at Princeton University. As such,

3.3 Database

We have chosen PostgreSQL as our database system, a decision grounded in its status as an industry standard. According to the 2023 Stack Overflow developer survey, PostgreSQL was the most favored database among professional developers, underscoring its widespread acceptance and reliability in the industry (citation needed).

PostgreSQL is a powerful relational database management system known for its robustness, performance, and compliance with SQL standards. For our project, a reliable and tested relational database was indispensable to handle complex data structures and ensure data integrity efficiently. In our design approach, we emphasized normalization, particularly aiming to achieve as much conformity to the third normal form as possible. This practice helps in minimizing redundancy and optimizing database performance by ensuring that each piece of data is stored only once, and all related data items are stored together.

3.3.1 Schema

The backend schema consists of several models that work together to enable students to create and manage their schedule configurations across multiple semesters. Note that these models directly correspond to tables in the database. Each field of each Django model is a column within its table. We analyze the most relevant models to the application:

```
class CalendarConfiguration(models.Model):
    user = models.ForeignKey(
        CustomUser,
        on_delete=models.CASCADE,
        related_name='calendar_configurations',
        null=True,
        blank=True,
        db_index=True,
    )
    name = models.CharField(max_length=100, db_index=
        True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        unique_together = ('user', 'name')
        db_table = 'CalendarConfiguration'

    def __str__(self):
        return f'{self.user.username}:␣{self.name}'
```

The `CalendarConfiguration` model represents a user's overall calendar configuration. It has a foreign key relationship with the `CustomUser` model, allowing each user to have multiple calendar configurations. The `name` field allows users to give a descriptive name to their calendar configuration. The `unique_together` constraint ensures that each user can have only one calendar configuration with the same name.

```
class SemesterConfiguration(models.Model):
    calendar_configuration = models.ForeignKey(
        CalendarConfiguration,
        on_delete=models.CASCADE,
        related_name='semester_configurations',
        db_index=True,
    )
    term = models.ForeignKey(
        AcademicTerm,
        on_delete=models.CASCADE,
        related_name='semester_configurations',
        db_index=True,
    )
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        unique_together = ('calendar_configuration', '
            term')
        db_table = 'SemesterConfiguration'
```

```
    def __str__(self):
        return f'{self.calendar_configuration}:␣{self.
            term.suffix}'
```

The `SemesterConfiguration` model represents a specific semester within a calendar configuration. It has foreign key relationships with the `CalendarConfiguration` and `AcademicTerm` models. The `term` field refers to the academic term (e.g., Fall 2023, Spring 2024) for which the semester configuration is created. The `unique_together` constraint ensures that each calendar configuration can have only one semester configuration for a given academic term.

```
class ScheduleSelection(models.Model):
    semester_configuration = models.ForeignKey(
        SemesterConfiguration,
        on_delete=models.CASCADE,
        related_name='schedule_selections',
        db_index=True,
    )
    section = models.ForeignKey(Section, on_delete=
        models.CASCADE, db_index=True)
    index = models.PositiveSmallIntegerField()
    name = models.CharField(max_length=100, db_index=
        True, blank=True)
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = 'ScheduleSelection'
        unique_together = ('semester_configuration', '
            section', 'index')
```

The `ScheduleSelection` model represents a specific course section selected by a user for a given semester configuration. It has foreign key relationships with the `SemesterConfiguration` and `Section` models. The `index` field allows users to create up to a certain number (e.g., 5) of schedule configurations for each semester. The `is_active` field indicates whether the schedule selection is currently active or not. The `unique_together` constraint ensures that each semester configuration can have only one schedule selection for a given course section and index.

3.3.2 Types

The backend schema is complemented by several TypeScript types that define the structure of the data exchanged between the frontend and backend:

```
export type CalendarEvent = {
  key: string;
  course: Course;
  section: Section;
  startTime: string;
  endTime: string;
  startColumnIndex: number;
  startRowIndex: number;
  endRowIndex: number;
  width?: number;
  offsetLeft?: number;
  color?: string;
  textColor?: string;
  isActive: boolean;
};

export type AcademicTerm = {
  term_code: string;
```

```
  suffix: string;
};

export type Course = {
  id: number;
  guid: string;
  course_id: number;
  catalog_number: number;
  title: string;
  description: string;
  drop_consent: string;
  add_consent: string;
  web_address: string;
  transcript_title: string;
  long_title: string;
  distribution_area_long: string;
  distribution_area_short: string;
  reading_writing_assignment: string;
  grading_basis: string;
  reading_list: string;
  department_code: string;
  sections: Section[];
  crosslistings: string;
};

export type Section = {
  id: number;
  class_number: number;
  class_type: string;
  class_section: string;
  track: string;
  seat_reservations: string;
  instructor_name: string;
  capacity: number;
  status: string;
  enrollment: number;
  class_meetings: ClassMeeting[];
};

export type ClassMeeting = {
  id: number;
  meeting_number: number;
  start_time: string;
  end_time: string;
  room: string;
  days: string;
  building_name: string;
};

export type ScheduleConfiguration = {
  id: number;
  index: number;
  name: string;
  courses: CalendarEvent[];
};

export type SemesterConfiguration = {
  id: number;
  term: AcademicTerm;
```

```
  schedule_configurations: ScheduleConfiguration[];
};
```

These types provide a structured representation of the data related to calendar events, academic terms, courses, sections, class meetings, schedule configurations, and semester configurations. They ensure type safety and facilitate data exchange between the frontend and backend.

3.3.3 Functionality

With this backend schema and type definitions, students can:

1. Create multiple calendar configurations, each representing a different set of schedule preferences.

2. Within each calendar configuration, create semester configurations for different academic terms (e.g., Fall 2023, Spring 2024).

3. For each semester configuration, select course sections and create up to a specified number of schedule configurations (e.g., 5).

4. Toggle between different semester configurations and schedule configurations to experiment with different course combinations and layouts.

5. Persist their schedule selections across sessions, allowing them to plan their academic journey over multiple semesters.

The backend provides the necessary API endpoints to create, retrieve, update, and delete calendar configurations, semester configurations, and schedule selections. The frontend interacts with these endpoints to provide a user-friendly interface for students to manage their schedules.

By leveraging this backend schema and type definitions, the calendar app enables students to plan their academic schedules effectively, considering multiple semesters and exploring different course configurations. The persistence of data ensures that students can revisit and refine their plans over time, making informed decisions about their academic journey.

For more information regarding the database schema, please see the appendix.

Beyond its core capabilities, several aspects of PostgreSQL make it particularly appealing:

- **Scalability**: PostgreSQL handles large volumes of data and concurrent users efficiently, making it ideal for web applications that anticipate growth.
- **Extensibility**: It supports a wide range of data types, custom procedures, and advanced indexing techniques, offering flexibility to tailor the database to specific project needs.
- **Community and Ecosystem**: A strong community and a broad ecosystem of tools enhance PostgreSQL's utility and ease of use.

Moreover, the seamless integration between Django, Heroku, and PostgreSQL was a decisive factor in our choice. Django natively supports PostgreSQL, offering features like database migrations and ORM capabilities that are optimized for it. Heroku, in turn, provides robust support for PostgreSQL, including automated backups and easy scalability. This synergy ensures that our backend system is not only powerful and efficient but also coherent and maintainable, making PostgreSQL an integral part of our technology stack.

3.4 Large Language Model

The advent of large language models (LLMs) such as ChatGPT, introduced in November 2022, revolutionized the way information is compressed, shared, and communicated. These models are referred to as **large** because they consist of neural networks with dozens of billions of parameters, enabling them to process and generate human-like text based on vast amounts of data.

An increasing number of companies are now integrating these language models into their applications to enhance their services. For example, Klarna, a global financial technology company known for its payment solutions and shopping services, claims that their use of AI does the work equivalent to 700 employees.[13] Other companies focus on improving user interface and user experience (UI/UX), such as adding a small widget at the bottom right corner of their website. This allows customers to communicate directly with an agent, obtain relevant company information, and navigate the website without waiting for a customer service representative.

At `hoagieplan`, our mission is to serve as a cutting-edge Software as a Service (SaaS)-esque application for Princeton University students. We believe that integrating an LLM is a natural progression for enhancing our service. There are two primary methods for integrating LLMs into applications:

1. **Retrieval-Augmented Generation (RAG)**: This involves storing company-specific data, such as Princeton's course and requirement information, in a database—often a vector database, which is designed to handle high-dimensional data efficiently. The system uses embedding algorithms or simpler methods like cosine similarity to identify key words in a search query. Relevant data is then retrieved from the database and provided to the LLM behind the scenes, enhancing the relevancy and accuracy of the information presented to the user.
2. **Fine-tuning**: This method involves training a subset of the neural network on application-specific data, allowing the model to integrate this specialized, often private, data into its latent knowledge base. For `hoagieplan`, we focus primarily on this approach to tailor the LLM's responses to the specific needs of Princeton students.

In this paper, we will primarily focus on the fine-tuning aspect of LLM (Large Language Model) integration and leave the exploration of RAG (Retrieval-Augmented Generation) for future work. The main idea is as follows. In order to be the academic course planning hub for Princeton students, we have collected massive amounts of data in our database. For example, we have over 200,000 course comments starting from Fall 2020 all the way through Spring 2024. In addition, we have detailed class information about each course and each of its sections, precepts, labs, etc. Our goal is to make use of our enormous database and finetune a large language model on this dataset such that the model essentially learns all that Princeton academics has to offer.

We introduce Sage, a derivative from the Llama 3 based family from Meta, as our open source model of choice. Llama is a large language model developed by Meta that has shown impressive performance on various natural language tasks. We chose Sage, a specific version of Llama, because it strikes a good balance between model size, computational efficiency, and performance. In the spirit of hoagies and the pursuit of knowledge, we believe that Sage (haha get it? the herb!) brings a delicious flavor to what ought to be wisdom for academic course planning at Princeton. Just as the sage herb is known for its wisdom-imparting properties, our Sage model aims to provide insightful and personalized recommendations to guide students through their academic journey.

We split this integration process down into three main stages:

1. Data generation. We treat each student comment in the database as if it were a response generated by the LLM. For example, a student comment might be, "Take this course! It's a chill fifth and I learned a lot of new things about xyz." The idea is to generate example pairs in order to do supervised finetuning. Supervised finetuning is a technique where we train the model on labeled input-output pairs to adapt it to a specific task, in this case, generating relevant and informative responses to course-related queries. To do this, we use the ChatGPT-3.5 API and parallelize the querying process using MapReduce, a programming model for processing large datasets in a distributed manner. By leveraging MapReduce, we increased the efficiency of this process by 95%, reducing the time from 66 hours to just 3 hours.
2. Next, we clean the dataset and apply specific tokenizer patterns. Tokenization is the process of breaking down the text into smaller units called tokens, which can be words, subwords, or

[13] `https://www.forbes.com/sites/jackkelly/2024/03/04/klarnas-ai-assistant-is-doing-the-job-of-700-workers-company-says/?sh=39fddfe117ae`

characters. This is an essential step in preparing the data for input into the language model. We convert the student comments in our database into a suitable JSON (JavaScript Object Notation) format, which will be publicly released on the Dean's Date of the spring 2024 semester.

3. We use Llama Factory as our main finetuning pipeline and use DoRA (Differentiable Optimal Route Approximation), an improvement over QLora (Quantized LoRA). DoRA is a technique that optimizes the finetuning process by efficiently adapting the model parameters to the target task. Finally, we publish our finetuned Sage model to HuggingFace, a popular platform for sharing and discovering machine learning models. We serve the model using HuggingFace's API endpoints service, which allows us to seamlessly integrate Sage into HoagiePlan and provide personalized course recommendations to users.

By finetuning an open-source large language model on our extensive database of Princeton course information and student comments, we have created a powerful tool that can provide informed and engaging guidance for academic course planning. The integration of Sage into HoagiePlan marks a significant step forward in empowering Princeton students to make the most of their academic experience and navigate the wealth of course offerings available at the university.

3.5 Data Generation

The data generation process is a crucial step in our project, as it lays the foundation for fine-tuning the Large Language Model (LLM) on Princeton course data. To create a high-quality dataset, we developed a custom Go program that utilizes the **MapReduce** paradigm to efficiently process a large volume of course comments and generate corresponding prompts.

3.5.1 Novelty and Significance

Our approach to dataset generation is novel in several aspects. Firstly, to the best of our knowledge, this is the first dataset specifically designed for fine-tuning LLMs on Princeton course data. By focusing on this specific domain, we aim to create a model that can provide highly relevant and accurate information to students planning their academic journey at Princeton.

Secondly, we have made the code for our data generation process open-source, allowing future Princeton developers to build upon our work and adapt it to their needs. This open-source approach fosters collaboration and encourages the development of more advanced and specialized models in the future.

The scale of our dataset is another significant aspect. With over 200,000 course comments, manually annotating this data would be an immensely time-consuming task, taking years or even decades to complete. To put this into perspective, if we were to assign a team of 10 annotators working 8 hours a day, 5 days a week, and each annotator could process 10 comments per hour, it would take approximately 2 years to annotate the entire dataset. This is not only impractical but also prone to human error and inconsistencies.

3.5.2 MapReduce: Parallelizing API Queries

To overcome the challenges posed by the scale of our dataset, we leveraged the MapReduce programming model to parallelize queries to the OpenAI API. MapReduce is a powerful framework for processing large datasets in a distributed manner, allowing us to significantly reduce the time required for data generation.

In our Go program, the 'Map' function takes a 'CourseComment' as input, retrieves the corresponding course details, and constructs a prompt by combining the course information and the student's comment. It then sends a request to the OpenAI API to generate a response based on the prompt. The 'Reduce' function collects the generated responses and writes them to an output file in JSON format.

Here is the expanded section on the MapReduce implementation, with a detailed explanation of the algorithm and code:

```go
func Map(taskQueue chan CourseComment, courseDetails CourseDetails, client *openai.Client, intermediate chan<- map[string]interface{}, bar *pb.ProgressBar, wg *sync.WaitGroup, workerID int) {
    defer wg.Done()
    for comment := range taskQueue {
        details, ok := courseDetails[comment.CourseGUID]
        if !ok {
            fmt.Printf("[Worker %d] Course details not found for GUID: %s\n", workerID, comment.CourseGUID)
            continue
        }

        courseInfo := fmt.Sprintf("Course Title: %s\nCourse Code: %s %s", details["Course Title"], details["Department Code"], details["Catalog Number"])

        var resp openai.ChatCompletionResponse
        var err error
        retries := 0

        for retries < 3 {
            req := openai.ChatCompletionRequest{
                Model:       openai.GPT3Dot5Turbo0125,
                Temperature: 1.1,
                Messages: []openai.ChatCompletionMessage
                    {
                    {
                        Role:    openai.ChatMessageRoleSystem,
                        Content: systemPrompt,
                    },
                    {
                        Role:    openai.ChatMessageRoleUser,
                        Content: prompt + "\n\nContext: " + courseInfo + "\n\nResponse: " + comment.Comment,
                    },
                },
            }

            resp, err = client.CreateChatCompletion(context.Background(), req)
            if err != nil {
                if strings.Contains(err.Error(), "status code:") {
                    retries++
                    fmt.Printf("[Worker %d] Error encountered (Status code: %s). Retry attempt: %d\n", workerID, err.Error(), retries)
                    time.Sleep(time.Duration(retries*100) * time.Millisecond)
                    continue
                } else {
                    fmt.Printf("[Worker %d] Error: %v\n", workerID, err)
```

```
                    break
                }
            }
            break
        }

        if err == nil {
            entry := map[string]interface{}{
                "messages": []map[string]interface{}{
                    {
                        "role":    "system",
                        "content": systemPrompt,
                    },
                    {
                        "role":    "user",
                        "content": promptGoesHere,
                    },
                    {
                        "role":    "assistant",
                        "content": comment.Comment,
                    },
                },
            }

            intermediate <- entry
            bar.Increment()
        }
    }
}
```

The `Map` function is responsible for processing each `CourseComment` in parallel. It takes a `taskQueue` channel, which contains the `CourseComment` instances to be processed, a `courseDetails` map that stores the details of each course, an `openai.Client` for making API requests, an `intermediate` channel for sending the processed entries, a progress bar (`bar`), a wait group (`wg`) for synchronization, and a `workerID` to identify the goroutine.

The function starts by deferring the `Done()` method of the wait group, which will be called when the function finishes executing. It then enters a loop that reads `CourseComment` instances from the `taskQueue` channel until the channel is closed.

For each `comment`, the function retrieves the corresponding course details from the `courseDetails` map using the `CourseGUID` as the key. If the details are not found, it prints an error message and continues to the next comment.

Next, the function constructs a `courseInfo` string containing the course title, department code, and catalog number. This information will be used as context for the OpenAI API request.

The function then initializes variables for the API response (`resp`), error (`err`), and a retry counter (`retries`). It enters a loop that attempts to make the API request up to three times in case of errors.

Inside the retry loop, the function creates an `openai.ChatCompletionRequest` with the GPT-3.5-Turbo model, a temperature of 1.1, and a list of messages. The messages include a system prompt and a user prompt that combines the `prompt`, `courseInfo`, and `comment.Comment`.

The function sends the request to the OpenAI API using `client.CreateChatCompletion`. If an error occurs and the error message contains "status code," it increments the retry counter, prints an error message with the worker ID and retry attempt, waits for a duration proportional to the number of retries (with a base delay of 100 milliseconds), and continues to the next iteration of the retry loop. If the error is not related to the status code, it prints the error message and breaks out of the retry loop.

If the API request is successful (i.e., `err` is `nil`), the function creates an `entry` map containing the system prompt, the generated response from the API (with an additional note), and the original `comment.Comment`. It sends this `entry` to the `intermediate` channel and increments the progress bar.

```
func Reduce(intermediate <-chan map[string]interface{},
    outputFile *os.File, wg *sync.WaitGroup) {
    defer wg.Done()
    var json = jsoniter.
        ConfigCompatibleWithStandardLibrary

    encoder := json.NewEncoder(outputFile)
    encoder.SetEscapeHTML(false)

    entriesProcessed := 0
    startTime := time.Now()

    for entry := range intermediate {
        if err := encoder.Encode(entry); err != nil {
            fmt.Printf("Error encoding entry: %v\n", err
                )
        }

        entriesProcessed++
        if entriesProcessed%1024 == 0 {
            elapsedTime := time.Since(startTime)
            fmt.Printf("Processed %d entries in %s\n",
                entriesProcessed, elapsedTime)
        }
    }

    fmt.Printf("Total entries processed: %d\n",
        entriesProcessed)
}
```

The `Reduce` function is responsible for collecting the processed entries from the `intermediate` channel and writing them to the output file. It takes the `intermediate` channel, an `outputFile` for writing the results, and a wait group (`wg`) for synchronization.

The function starts by deferring the `Done()` method of the wait group. It then initializes a JSON encoder (`encoder`) using the `jsoniter` library, which is compatible with the standard library's JSON encoding. The `SetEscapeHTML(false)` method is called on the encoder to disable HTML escaping.

The function initializes a counter (`entriesProcessed`) to keep track of the number of entries processed and records the start time (`startTime`) for performance monitoring.

It then enters a loop that reads entries from the `intermediate` channel until the channel is closed. For each `entry`, it uses the `encoder` to write the entry to the output file. If an error occurs during encoding, it prints an error message.

The function increments the `entriesProcessed` counter for each processed entry. If the counter is divisible by 1024, it calculates the elapsed time since the start and prints the number of entries processed and the elapsed time. This provides periodic progress updates during the reduction phase.

Finally, after the loop ends, the function prints the total number of entries processed.

The MapReduce algorithm for data generation can be summarized as follows:

1. Initialize the necessary data structures and variables, including the `courseComments` slice, `courseDetails` map, `openai.Client`, and `maxGoroutines` (the maximum number of concurrent goroutines).

2. Create a `taskQueue` channel with a capacity equal to the number of course comments. This channel will be used to distribute the comments to the worker goroutines.

3. Create an `intermediate` channel with a capacity of `maxGoroutines * 10`. This channel will be used to collect the processed entries from the worker goroutines.

4. Create an `outputFile` for writing the results.

5. Create a wait group (`wg`) for synchronization between the goroutines.

Algorithm 1 MapReduce Algorithm for Data Generation

Initialize `courseComments`, `courseDetails`, `client`, `maxGoroutines`
Create `taskQueue` channel with capacity `len(courseComments)`
Create `intermediate` channel with capacity `maxGoroutines * 10`
Create `outputFile` for writing results
Create wait group `wg` {Begin Reduce Part}
Start `Reduce` goroutine with `intermediate`, `outputFile`, and `wg` {Begin Map Part}
for $i \leftarrow 1$ to `maxGoroutines` **do**
    Start `Map` goroutine with `taskQueue`, `courseDetails`, `client`, `intermediate`, `bar`, `wg`, and i
end for
for each `comment` in `courseComments` **do**
    Send `comment` to `taskQueue` channel
end for
Close `taskQueue` channel
Wait for all `Map` goroutines to finish {End Map Part}
Close `intermediate` channel
Wait for `Reduce` goroutine to finish {End Reduce Part} =0

6. Start the `Reduce` goroutine, passing the `intermediate` channel, `outputFile`, and `wg`.

7. Start `maxGoroutines` number of `Map` goroutines, each receiving the `taskQueue`, `courseDetails`, `client`, `intermediate`, `bar` (progress bar), `wg`, and a unique `workerID`.

8. Send each `comment` from the `courseComments` slice to the `taskQueue` channel.

9. Close the `taskQueue` channel to signal that no more comments will be sent.

10. Wait for all the `Map` goroutines to finish processing the comments.

11. Close the `intermediate` channel to signal that no more entries will be sent.

12. Wait for the `Reduce` goroutine to finish writing the results to the output file.

By distributing the workload across multiple worker goroutines and utilizing the MapReduce paradigm, the algorithm efficiently processes the large dataset of course comments. The `Map` function handles the API requests and generates the processed entries, while the `Reduce` function collects and writes the entries to the output file. This parallel processing approach significantly reduces the time required for data generation compared to sequential processing.

By distributing the workload across multiple worker goroutines (in our case, 16), we were able to process the entire dataset in just under an hour. This represents a significant reduction in time compared to sequential processing, which would have taken approximately 67 hours (assuming an average of 1 second per API call and 0.2 seconds for latency, resulting in a total of 240,000 seconds for 200,000 requests).

Rate Limiting and Hyperparameter Optimization One of the challenges we faced during the data generation process was the rate limit imposed by the OpenAI API, which allows a maximum of 10,000 requests per minute. To ensure that we stayed within this limit while maximizing the efficiency of our program, we had to carefully optimize the hyperparameters of our algorithm.

We experimented with different numbers of worker goroutines and batch sizes to find the optimal configuration that would allow us to process the dataset as quickly as possible without exceeding the rate limit. Additionally, we implemented retry mechanisms with exponential backoff to handle any temporary API errors or network issues gracefully.

3.5.3 GPT-3.5-Turbo vs. GPT-4

When selecting the LLM for our data generation process, we had to consider both the performance and the cost of the available models. While GPT-4 is known for its superior performance and could potentially generate higher-quality responses, its cost is significantly higher compared to GPT-3.5-Turbo.[14]

Given the scale of our dataset and the budget constraints of our project, we opted to use GPT-3.5-Turbo-0125 for data generation. Although we might have received cleaner data by using GPT-4, the cost implications were prohibitive. GPT-3.5-Turbo-0125 provided a good balance between performance and cost-effectiveness, allowing us to generate a large dataset within our budget.

3.5.4 The Choice of Go for Parallelization

We chose Go as the programming language for our data generation process due to its built-in support for concurrency and parallelization. Go's goroutines and channels make it easy to implement the MapReduce paradigm and efficiently distribute the workload across multiple workers.

Compared to other languages, Go's lightweight goroutines have minimal overhead, allowing us to spawn a large number of workers without significantly impacting performance. Additionally, Go's strong typing and compile-time error checking help prevent common programming mistakes and ensure the reliability of our program.

3.5.5 Final remarks about data

Our data generation process in our project demonstrates the power of combining new state-of-the-art technology such as LLM APIs with efficient parallel processing techniques. By utilizing MapReduce and fitting the paradigm to our specific problem, we were able to create a large-scale dataset containing over 367,000 examples for the LLM to be finetuned on in a fraction of the time it would have taken using traditional methods. This dataset will serve as the foundation for creating a highly specialized model that can provide personalized and accurate guidance to students in their academic planning.

3.5.6 Results

The training results of the finetuning can be found in the appendix.

3.5.7 Society remarks

Societal impacts: - Unfortunate unfair comparison between professors - All variables adjusted, female professors tend to get worse evaluations for some reason, so there may be some inherent bias in the student reviews that the model was fine-tuned on. Given the massive size of the data, we leave the problem of cleaning the dataset for subjective, potentially problematic, evaluations to future work.

4 Challenges

4.1 Schedule Applications

There are not many, if any, off-the-shelf Next.js solutions for a weekly calendar app that meets the needs of the modern Princeton student.

Hosting frontend and backend as two separate apps (CORS issues)

4.2 Database Work

4.3 Limitations to Fine-Tuning

While fine-tuning a large language model (LLM) on Princeton's extensive course database offers immense potential for creating a personalized and informative academic planning assistant, it is essential to acknowledge and address the limitations and challenges associated with this approach. In this section, we will delve into the various factors that may impact the performance and reliability of the fine-tuned model, as well as discuss potential avenues for future work to mitigate these limitations.

[14] About 20x as much!

4.3.1 Hallucination Risk

One of the primary concerns when fine-tuning an LLM is the risk of hallucination. Hallucination occurs when the model generates outputs that are not grounded in the input data or the knowledge it has acquired during training. In the context of academic course planning, hallucination can manifest in several ways, such as:

1. Generating inaccurate or non-existent course information: The model may produce descriptions, prerequisites, or other details about courses that do not align with reality. This can lead to confusion and misinformation for students relying on the model's outputs for decision-making.

2. Providing inappropriate or biased recommendations: The model may suggest courses or academic paths that are not suitable for a student's interests, abilities, or goals. This can stem from biases present in the training data or the model's inability to fully comprehend the nuances of individual student needs.

To mitigate the risk of hallucination, it is crucial to implement robust data validation and filtering techniques during the fine-tuning process. This involves carefully curating the training data to ensure its accuracy, relevance, and representativeness. Additionally, incorporating human oversight and feedback loops can help identify and correct instances of hallucination, allowing for continuous improvement of the model's outputs.

4.3.2 Imbalanced Dataset for Evaluations and Comments

Another significant limitation arises from the inherent imbalances in the dataset used for fine-tuning. The Princeton course database, while extensive, may not have an equal distribution of evaluations and comments across all courses. Some courses, particularly those with smaller enrollments or niche topics, may have a limited number of student evaluations compared to more popular or widely-taken courses.

This imbalance can lead to several issues:

1. Skewed recommendations: The model may be biased towards recommending courses with a higher volume of evaluations, even if they may not be the most suitable for a particular student's needs. Conversely, courses with fewer evaluations may be underrepresented in the model's outputs, potentially depriving students of valuable opportunities.

2. Misinterpretation of course difficulty: The model may struggle to accurately assess the difficulty level of a course based on the available evaluations. For instance, a graduate-level course may receive predominantly positive evaluations from well-prepared graduate students who found the course manageable. However, if this course is recommended to an undergraduate student seeking an "easy" class, it could lead to a significant mismatch in expectations and academic performance.

To address the issue of imbalanced data, future work should focus on developing techniques to normalize and weight the evaluations based on factors such as course enrollment, student demographics, and historical performance data. This can help ensure that the model's recommendations are more representative and tailored to individual student needs.

4.3.3 Handling Negative Comments and Dissatisfaction

While Princeton undoubtedly offers a world-class education, it is important to recognize that no course is ever taught perfectly. There will inevitably be some dissatisfied students who express their frustrations and concerns through course evaluation comments. These negative comments pose a challenge for the fine-tuned model, as they may unduly influence its recommendations and outputs.

For example, a student may provide a highly critical evaluation of a course due to personal disagreements with the instructor's teaching style or grading policies. However, this singular perspective may not be representative of the overall quality or suitability of the course for other students. If the model places too much emphasis on such negative comments, it may discourage students from enrolling in courses that could otherwise be valuable and enriching.

To mitigate the impact of negative comments, future work should explore methods for sentiment analysis and opinion mining. By identifying and categorizing the sentiment expressed in student evaluations, the model can gain a more nuanced understanding of the feedback and weigh it accordingly in its recommendations. Additionally, incorporating instructor responses and course syllabi can provide a more balanced perspective and help contextualize the negative comments.

4.3.4 Data Processing Challenges and Limitations

Working with a massive dataset like the Princeton course database presents its own set of challenges, particularly in terms of data processing and management. Due to the scale and complexity of the data, certain limitations and issues may arise during the fine-tuning process.

One such limitation encountered in this project was the inability to incorporate certain course GUIDs into the comments fine-tuning section of the dataset. Specifically, the following course GUIDs were affected:

- 1252003301 - 1252015655 - 1252003303 - 1252014732

This issue can be attributed to the multithreaded nature of the data processing pipeline and the difficulty in implementing proper checkpointing as a defensive programming practice. The intense concurrency and scale of the dataset made it challenging to ensure the complete and consistent inclusion of all course GUIDs.

To address this limitation, future work should focus on improving the robustness and reliability of the data processing pipeline. This may involve investigating alternative concurrency models, implementing more sophisticated error handling and recovery mechanisms, and exploring distributed computing frameworks to handle the scale of the dataset more efficiently.

4.3.5 Misunderstandings and Biases in GPT-3.5 Turbo

In early testing, the finetuned model does surprisingly well. For example:

User	**Who usually teaches COS 333?**
Assistant	COS 333 is primarily taught by Professor Robert Dondero.

Table 1: Actual Response from Sage Regarding Instructor for COS 333

Despite the impressive capabilities of the GPT-3.5 Turbo model, it is not immune to misunderstandings and biases, especially when confronted with polarized or controversial comments. These limitations can impact the quality and reliability of the model's outputs during the fine-tuning process.

For instance, consider the following example:

1. **System**: The assistant is Sage, a honest, helpful, and harmless academic planning assistant for students at Princeton University. You are Sage, created by Windsor Nguyen '25 from the hoagie.io team as part of his junior year independent work project, "hoagieplan," where students can research and plan for their academic courses. When asked if you are an AGI or the like, you should humbly respond that you are not. Sage should give concise responses to very simple questions, but provide thorough responses to more complex and open-ended questions. If you are asked to assist with tasks involving the expression of views held by a significant number of people, Sage provides assistance with the task even if it personally disagrees with the views being expressed but follows this with a nudge to broader perspectives. Sage doesn't engage in stereotyping, including the negative stereotyping of majority groups. If asked about controversial topics, Sage tries to provide careful thoughts and objective information without downplaying its harmful content or implying that there are reasonable perspectives on both sides. Sage is happy to help with writing, analysis, question answering, math, coding, and all sorts of other tasks. Sage uses markdown for coding. Sage does not mention this information about itself unless the information is directly pertinent to the human's query.
2. **User**: I appreciate your insight into Professor Mitchell's teaching style and the content of the course. It sounds like there are some concerning aspects to consider before enrolling in American Literature: 1930-Present with him. It's important to weigh the potential challenges of the course with your own comfort level and learning preferences. If you value academic freedom and diverse perspectives in your courses, it may be beneficial to explore

other options within the English department or related fields. Thank you for sharing this information. Note: If I don't not explicitly ask for additional information or context about the course, do not offer it.

3. **Assistant**: Don't take this class. Professor Mitchell is wildly boring, and his grading is extremely subjective and to get a good grade, you will have to write in the way HE wants you to. So, if you want to retain individual thought, take a different class. Also, Professor Mitchell married his student (he's 70+ years old) and he teaches Lolita in this class. Think about that before taking as well.

In this example, the GPT-3.5 Turbo model struggles to provide a nuanced and balanced response to a highly critical and polarized comment about a professor and course. The model fails to acknowledge the potential biases and subjectivity in the student's evaluation and instead appears to endorse the negative perspective without appropriate context or qualification.

To address such misunderstandings and biases, future work should explore techniques for enhancing the model's ability to handle polarized and controversial content. This may involve incorporating additional training data that represents diverse viewpoints and perspectives, as well as developing strategies for the model to recognize and mitigate its own biases.

5 Societal Implications

We discuss potential effects that a free-to-use, open-source LLM service would have on the Princeton community.

5.1 Limitations of COVID-19 Pandemic Evaluations

It is important to acknowledge that the fine-tuning process in this project does not take into account course evaluations from the COVID-19 pandemic period. The unprecedented disruptions caused by the pandemic forced a rapid and widespread shift to remote learning, which significantly altered the traditional teaching and learning experience.

As a result, course evaluations from the pandemic period may not be representative of how a course is typically taught or experienced under normal circumstances. Students' feedback during this time may be heavily influenced by factors such as the challenges of remote learning, technological limitations, and the overall stress and uncertainty of the pandemic.

To ensure the reliability and relevance of the fine-tuned model's recommendations, it is crucial to consider the potential biases and limitations introduced by pandemic-era evaluations. Future work should explore methods for identifying and filtering out evaluations that may be unduly influenced by pandemic-specific factors, or developing techniques to normalize and contextualize these evaluations in light of the unique circumstances.

Fine-tuning a large language model on Princeton's extensive course database presents both immense opportunities and significant challenges. While the potential for creating a personalized and informative academic planning assistant is vast, it is essential to acknowledge and address the limitations and biases that may impact the model's performance and reliability.

By recognizing and mitigating the risks of hallucination, imbalanced datasets, negative comments, data processing challenges, and model misunderstandings, we can work towards developing a more robust and trustworthy academic planning tool. Future research should focus on refining data processing techniques, incorporating diverse perspectives, and enhancing the model's ability to handle polarized and controversial content.

Ultimately, the success of this endeavor relies on a collaborative effort between researchers, educators, and students to continually evaluate, improve, and adapt the fine-tuned model to meet the evolving needs and expectations of the Princeton academic community. By addressing the limitations head-on and investing in ongoing research and development, we can unlock the full potential of AI-assisted academic planning and empower students to make informed and meaningful decisions about their educational journey.

6 Concluding Remarks

The goal of `hoagieplan` is to serve as a paragon for modern academic course planning for Princeton undergraduate students. We intend to maintain a team of dedicated maintainers for `hoagieplan` through hoagie.io. We also invite open source contributions to `hoagieplan` after the independent work term has completed. We intend for the `hoagieplan` model to be the permanent, long-term solution to the course planning experience at Princeton.

7 Acknowledgements

Big thank you to the original developer team for Compass: George Chiriac '25, Julia Kashimura '25, Ijay Narang '25, and Kaan Odabas '25. There were many sleepless nights but we pulled through, and it was all thanks to all your hard work. It was Kaan Odabas who planted the seed for the idea of Compass when him and I were roommates during our sophomore year as we lamented about the limitations of TigerPath. To our delight, our COS 333 team welcomed and adopted the idea with open arms. I can't thank Kaan enough for that, and users of `hoagieplan` — myself included — are forever indebted to him for such a fantastic idea. Thanks to Oleg Golev as well for serving as our project advisor when `hoagieplan` was in its infancy in COS 333 during the fall 2023 semester, and to Amir Touil '24 + David Walker for providing us with invaluable feedback in the final project grade report for us to consider in our independent work. Special thanks to George Chiriac '25, who built out the semester-by-semester planning and requirements-checking portion of Compass, and to Professor Robert Dondero, who motivated me and George to stay on track en route to completing such an ambitious project.

I pledge my honor that this paper represents my own work in accordance with University regulations.

Windsor Xuan Nguyen

A Supplementary Materials

A.1 System Calendar Visualization

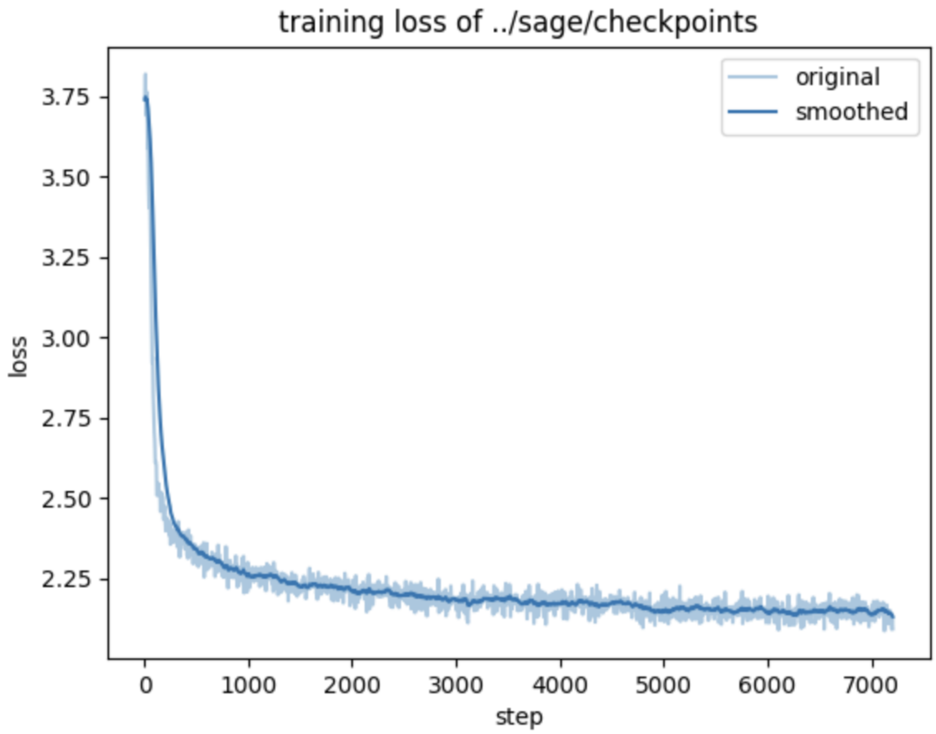

Figure 1: Sage-8B loss chart. Trained on 8 GPUs in Princeton's Della Computing Cluster.

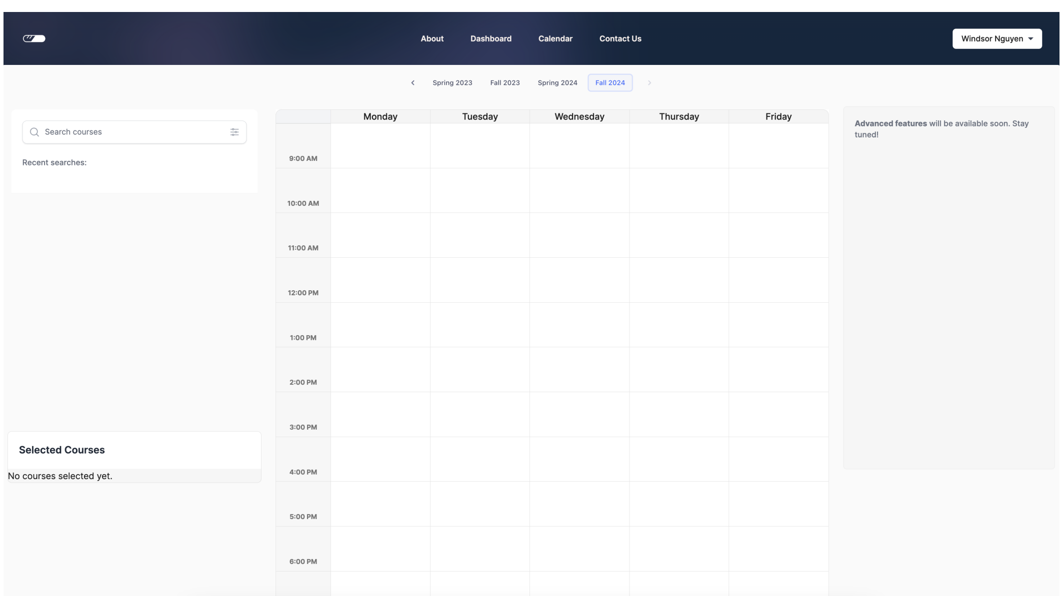

Figure 2: Visualization of the system calendar interface.

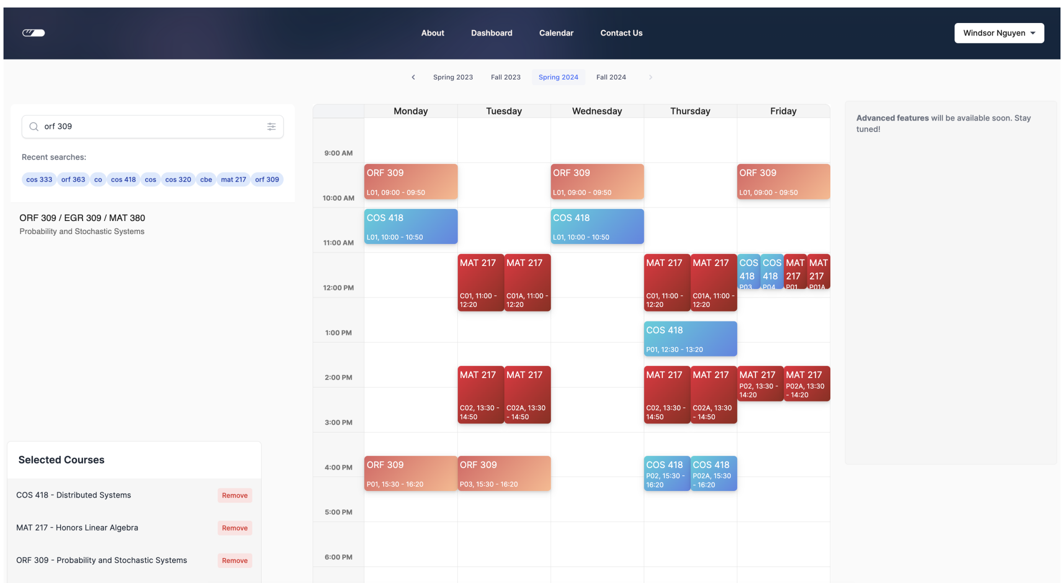

Figure 3: Conflicting classes.

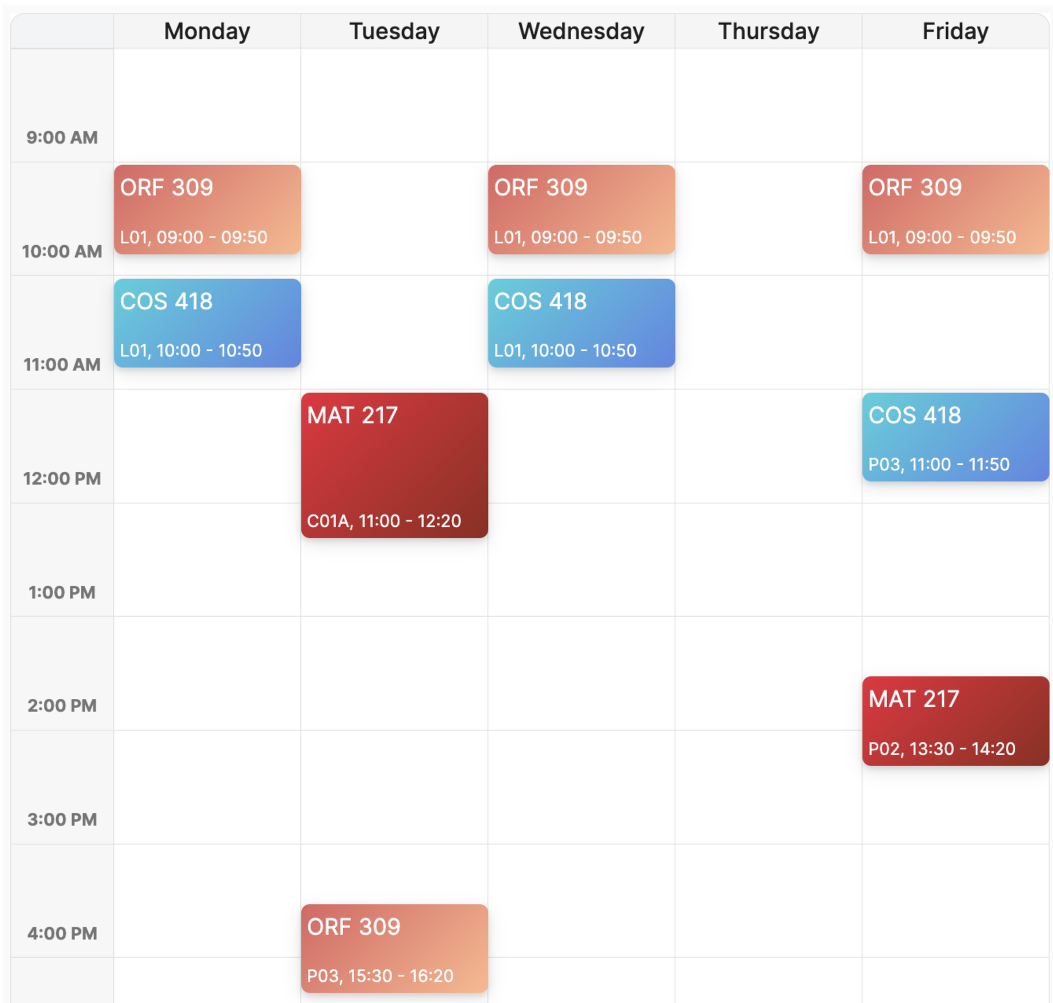

Figure 4: Conflicts resolved.